

BEST AVAILABLE COPY

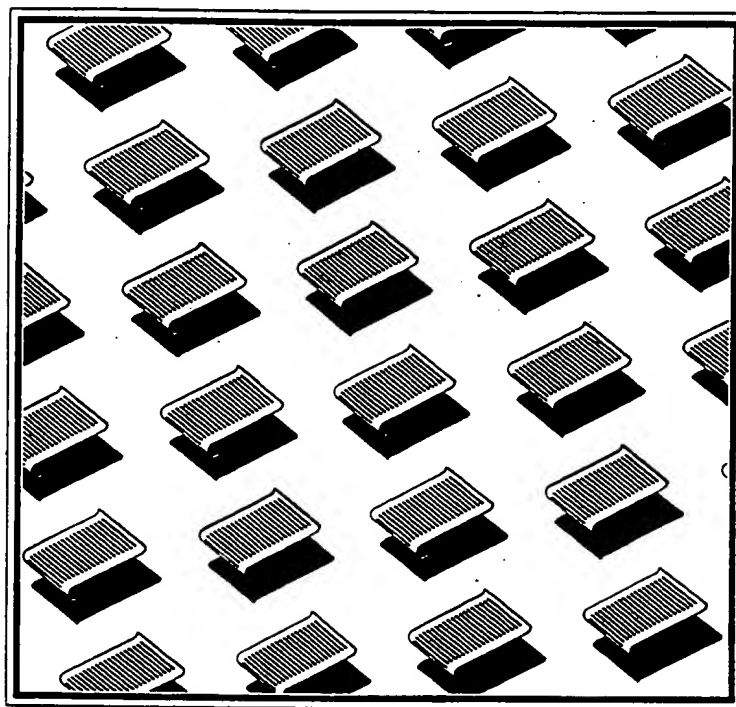
APPENDIX 1

© Copyright 1995
Pure Software, Inc.
All Rights Reserved

009207-102600

EMPOWER

Script Development



f o r E M P O W E R / C S



PERFORMIX

09597994, 100500

LIMITATION OF LIABILITY

PERFORMIX makes no warranty or representation of any kind, either expressed or implied, with respect to this software, updates, or documentation, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. This software, updates, and documentation are provided 'AS IS'. The entire risk as to their quality, performance, and results is assumed by you. Some states do not allow the exclusion of implied warranties, so the above limitation may not apply to you.

In no event will PERFORMIX be liable to you for any damages whatsoever (including but not limited to direct, indirect, special, incidental, or consequential damages) arising out of the use or inability to use the software, updates, or documentation even if PERFORMIX has been advised of the possibility of such damages. In particular, PERFORMIX is not responsible for any damages including but not limited to those resulting from lost profits or revenue, loss of use to the computer program, loss of data, claims by third parties, or for other similar damages. Some states do not allow the execution or limitation of liability for consequential or incidental damages, so the above limitation may not apply to you.

COPYRIGHT

The PERFORMIX documentation and the software are copyrighted and protected by both the Universal Copyright Convention and the Berne Convention. All rights are reserved. No part of this documentation nor the software may be copied, reproduced, translated, or transmitted in any form or by any means except as expressly described in your license agreement.

U.S. GOVERNMENT RESTRICTED RIGHTS

If this software and documentation are acquired by or on behalf of a unit or agency of the United States Government this provision applies. This software and documentation: (a) were developed at private expense, and no part of it was developed with government funds, (b) are a trade secret of PERFORMIX, Inc. for all purposes of the Freedom of Information Act, (c) are "commercial computer software" and "computer software documentation" subject to limited utilization as provided in the contract between the vendor and the government entity, and (d) in all respects are proprietary data belonging solely to PERFORMIX, Inc.

The enclosed software and documentation are provided with RESTRICTED AND LIMITED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR 52.227-14 (June 1987) Alternate III(g)(3)(June 1987), FAR 52.227-19(June 1987), or DFARS 252.227-7013 (c)(1)(ii)(October 1988), as applicable. Contractor/Manufacturer is PERFORMIX, Inc., 8200 Greensboro Drive, Suite 1475, McLean, VA 22102. Unpublished-rights reserved under the copyright laws of the United States.

EMPOWER/CS is a trademark of PERFORMIX, Inc. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows is a trademark of Microsoft Corporation. IBM is a registered trademark of International Business Machines Corporation. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. Oracle is a registered trademark of Oracle Corporation. Sybase is a registered trademark of Sybase, Inc. All other trademarks are the property of their respective holders.

Software Version 1.0.1, User's Guide Version 1.0.1



PERFORMIX, Inc.
8200 Greensboro Drive, Suite 1475
McLean, VA 22102
(703) 448-6606 (phone)
(703) 893-1939 (fax)

009697994 "102600

Table of Contents

1.0 Welcome to EMPOWER/CS	1-1
1.1 Organization of the EMPOWER User's Guides	1-1
1.2 Organization of this Manual	1-2
1.3 User's Guide Conventions	1-3
 2.0 Introduction to Load Testing with EMPOWER/CS	 2-1
2.1 Why Use EMPOWER/CS?	2-1
2.1.1 Applications Development	2-1
2.1.2 Computer Purchase Decisions	2-2
2.1.3 Capacity Planning	2-2
2.1.4 Product Demonstrations	2-3
2.2 How Does EMPOWER/CS Work?	2-3
2.3 Testing Process	2-5
2.4 EMPOWER/CS Tools	2-6
2.4.1 Capture	2-7
2.4.2 Csccl	2-10
2.4.3 Mix	2-11
2.4.4 Reports	2-12
2.4.5 Draw	2-12
2.4.6 Monitor	2-14
2.4.7 Global Variables	2-14
 3.0 Installation	 3-1
3.1 The EMPOWER/CS Environment	3-2
3.2 Installing the User PC, the UNIX Script Driver, and the Database Server	3-2

3.3	Installing the EMPOWER/CS Software	3-2
3.3.1	Installing EMPOWER/CS on the UNIX Script Driver	3-3
3.3.2	Configuring the UNIX Driver for EMPOWER/CS	3-5
3.3.3	Installing EMPOWER/CS on the PC	3-6
3.4	Before Starting Capture	3-7

4.0	Capture	4-1
4.1	Suggestions for Executing Capture	4-3
4.2	Executing Capture	4-4
4.2.1	Options	4-7
4.2.1.1	Think Threshold	4-9
4.2.1.2	Bring to Front with	4-10
4.2.1.3	View Script	4-10
4.2.1.4	Database Traffic Only	4-11
4.2.1.5	Insert Timer	4-11
4.2.1.6	Database Chooser	4-12
4.2.1.7	Transfer Script after Capture	4-13
4.2.1.8	License Daemon	4-15
4.2.2	Directory	4-15
4.2.3	Capturing Application Activity into a Script File	4-16
4.2.4	Comments, Functions, and Timers	4-18
4.2.4.1	Inserting Timers	4-19
4.2.4.2	Inserting Functions	4-19
4.2.4.3	Comments	4-22
4.2.5	Completing Your Capture Session	4-22

5.0	Csccl	5-1
5.1	Csccl Syntax	5-1
5.2	Csccl Environment Variables	5-2
5.3	Compiling with Csccl	5-4
5.3.1	Specifying the Binary Name with -o	5-4
5.3.2	Preprocessing the Source Script with -E	5-5
5.3.3	Modular Script Design	5-6
5.3.4	Extending Modular Script Compilation with -F	5-10

5.3.5	Automatic Creation of Function Archives.....	5-11
5.3.6	Optimizing and Stripping the Script Binary.....	5-12
5.3.7	Excluding Help Information from the Script Binary.....	5-12
5.3.8	Excluding Monitor Code.....	5-13
5.3.9	The Compiler Command Line.....	5-13
5.4	Cscc Compilation Messages.....	5-14

6.0	Script Execution	6-1
6.1	Non-Display Mode	6-1
6.2	Script Execution in Display Mode.....	6-2
6.3	Script Execution Options:.....	6-5
6.3.1	-d.....	6-5
6.3.2	Changing the Script ID.....	6-5
6.3.3	Specifying a Log File.....	6-6
6.3.4	Specifying Arguments.....	6-7
6.4	The Log File.....	6-8
6.5	What Comes Next?.....	6-12

7.0	Script Content and Enhancement.....	7-1
7.1	Script Content	7-1
7.1.1	General Script Functions.....	7-4
7.1.1.1	Begin/End Functions	7-4
7.1.1.2	Typerate and Pointerrate	7-5
7.1.1.3	Think Time Functions	7-7
7.1.1.4	Timeouts and Database Errors.....	7-9
7.1.1.5	Set, Unset	7-11
7.1.1.6	Mouse Activity	7-12
7.1.1.7	Keyboard Activity	7-14
7.1.1.8	Type.....	7-17
7.1.1.9	ButtonPush	7-19
7.1.1.10	InitialWindow	7-20
7.1.1.11	AppWait Delay	7-22
7.1.1.12	WindowRcv	7-23
7.1.1.13	CurrentWindow	7-24

Discipline in the Classroom

7.1.2	Interacting with the SUT	7-25
7.1.2.1	Communication Structures	7-25
7.1.2.2	Procedure for Interacting with the SUT	7-27
7.1.2.3	Open an Environment	7-30
7.1.2.4	Connect to the SUT	7-31
7.1.2.5	Open the Cursors	7-32
7.1.2.6	Parse the SQL Statement	7-32
7.1.2.7	Describe Select-List Items	7-34
7.1.2.8	Bind the Addresses of Input Variables	7-35
7.1.2.9	Define	7-36
7.1.2.10	Execute the Parse	7-37
7.1.2.11	Fetch the Rows of Data for a Query	7-38
7.1.2.12	Close the Cursor Structures	7-40
7.1.2.13	Disconnect from the SUT	7-40
7.1.2.14	Close the Environment	7-40
7.1.3	Other Database Functions	7-41
7.1.3.1	Dbset	7-41
7.1.3.2	Data	7-45
7.1.3.3	SqlExec	7-47
7.1.3.4	Commit, Rollback	7-47
7.1.4	Editing Database Functions	7-48
7.1.5	Comments	7-49
7.2	Data Types	7-49
7.3	Script Arguments	7-51
7.3.1	Argument Syntax	7-51
7.3.2	Argument Examples	7-52
7.4	Script Variables	7-53
7.5	Editing Your Scripts	7-55
7.5.1	Recording Messages	7-55
7.5.2	File Input/Output Functions	7-56
7.5.2.1	Fioshare	7-57
7.5.2.2	Fiounshare	7-59
7.5.2.3	Fioopen	7-59
7.5.2.4	Fioclose	7-60
7.5.2.5	Fiodelimiter	7-60
7.5.2.6	Fioreadline	7-60
7.5.2.7	Fioreadfield	7-61
7.5.2.8	Fioreadfields	7-61

8.0	EMPOWER/CS Tools	8-1
8.1	XY	8-1
8.2	Modules	8-2
8.3	Transfer	8-3
8.4	Tree	8-6
	8.4.1 The Tree Structure	8-6
	8.4.2 Using the Tree Tool	8-8

1.0 Welcome to EMPOWER/CS

Welcome to EMPOWER/CS, PERFORMIX Inc.'s full-featured solution for efficient client/server load testing. With EMPOWER/CS, you can emulate multiple users from a single load testing machine to accurately measure response time and throughput of your client/server environment. EMPOWER/CS captures and replays actual client/server activities running from Microsoft Windows PCs. It stress tests applications and the database server by emulating multiple PCs that interact with the server and then measures client/server performance by summarizing response times.

Until now, client/server load testing required assembling individual PCs for each system-supported user, which could involve up to *hundreds* or *thousands* of PCs. With EMPOWER/CS, this costly and cumbersome process is reduced to **one** UNIX driver machine that emulates as many PCs needed to test your entire client/server environment.

You can use EMPOWER/CS to load test your client/server applications during the entire cycle of development, to determine your database server's performance under peak load conditions, and to help determine future needs of your client/server environment.

This User's Guide is designed to help you understand the testing processes and capabilities of EMPOWER/CS and to guide you through efficient client/server load testing.

1.1 Organization of the EMPOWER User's Guides

The complete documentation for EMPOWER/CS includes three user's manuals which contain general use information, installation instructions, technical reference material, and examples. The following list identifies each user manual:

Describes commands, functions, and possible error messages for EMPOWER/CS. This manual also includes information for contacting PERFORMIX technical support

Regular Font	Used for all regular body text
Arial Font	Represents elements of the MS Windows environment such as pushbuttons, window titles, user entries, etc.
Mono-spaced Font	Used for all command, function, and file names; for all examples; and, generally, for any computer generated text
Bold mono-spaced font	In examples, represents entries made by the EMPOWER/CS user
<code>{-S scriptid}</code>	In command syntaxes, text within these square brackets represents optional command parameters
<code>()</code>	Vertical lines separate command parameter choices
<code>...</code>	Within scripts, the ellipsis marks indicate that some script content was left out for brevity
<code>Endfunction()</code>	Parentheses are included with script functions mentioned in regular body text. For most functions, one or more parameters will be listed in the parentheses when the function is used in a script
<code>Parse()</code>	EMPOWER/CS script functions use initial capitalization
<code>csc</code>	EMPOWER/CS command names use all lowercase letters
Capture	When an EMPOWER/CS tool is mentioned within regular body text, it is shown in regular font with initial caps

The term, "SUT," refers to your client/server system under test. The client/server SUT includes the database server and the database(s) on that server used for your emulation.

009204 4662560

2.0 Introduction to Load Testing with EMPOWER/CS

EMPOWER/CS tests the performance of a client/server application or system by emulating application activity of multiple PC users and by gathering response time data. Using EMPOWER/CS eliminates the need for many PCs to stress test your client/server system under test (SUT) because you can emulate multiple PC users on a single UNIX driver machine applying a realistic load to the SUT. EMPOWER/CS tools allow you to capture actual user activity in a script file, combine and execute several scripts at once, and collect performance statistics.

2.1 Why Use EMPOWER/CS?

You can use EMPOWER/CS to support applications development, computer purchase decisions, capacity planning, and product demonstrations.

2.1.1 Applications Development

After you develop an application, most problems do not arise until your customer tries to use the application during peak load conditions. Load testing with EMPOWER/CS measures the true scalability and performance of new client/server applications and systems before they are introduced to actual users. By emulating multiple users with EMPOWER/CS, you can identify problems and defects from the very beginning of the development cycle up to new releases of the product.

Companies that do not load test their client/server products risk introducing applications to the marketplace that fail to meet critical user expectations.

2.1.2 Computer Purchase Decisions

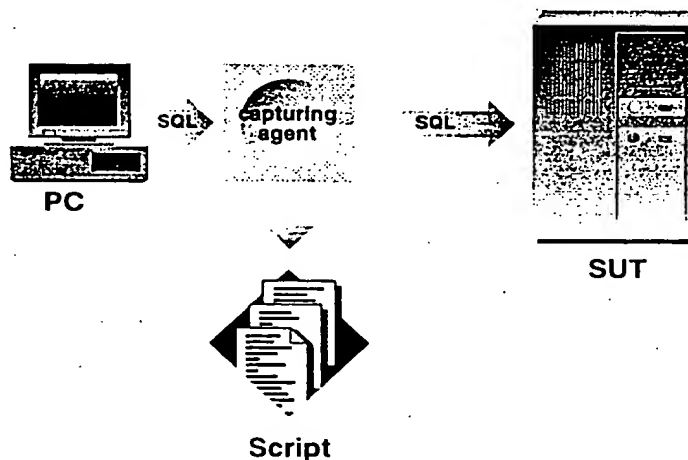
2.1.3 Capacity Planning

2.1.4 Product Demonstrations

EMPOWER/CS is highly effective in demonstrating your client/server product or application to potential customers. A performance test emulating actual user and database activity provides your customers with a feeling of confidence that your client/server application will perform in the field as promised. This is especially useful if you do not have reference sites for your potential customers' workload.

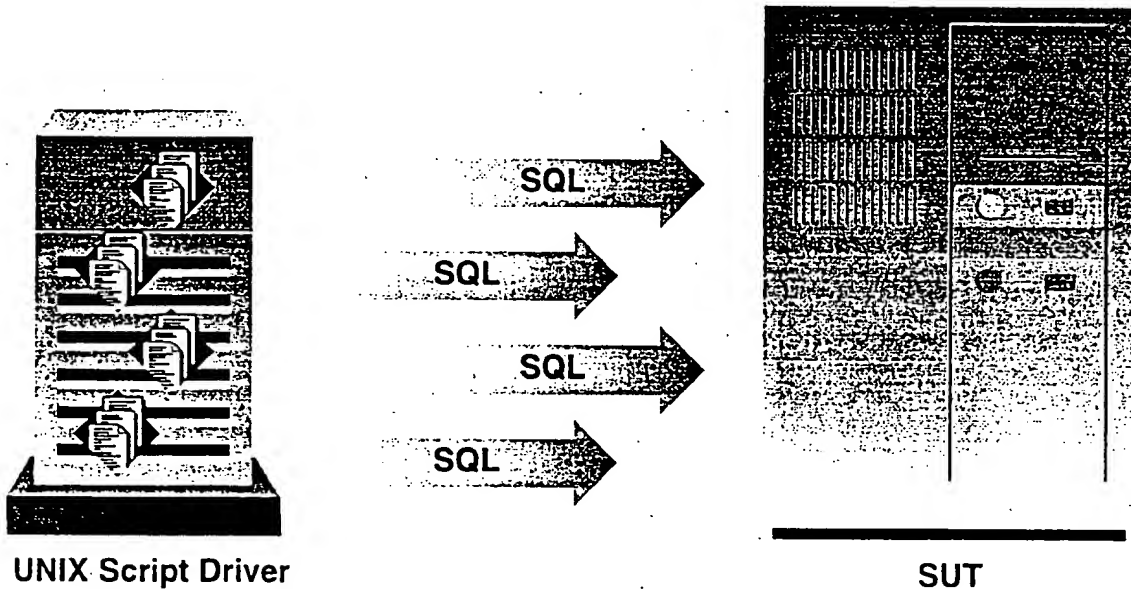
2.2 How Does EMPOWER/CS Work?

EMPOWER/CS places a capturing agent on a Microsoft Windows PC that records dialog between the PC and the SUT. From this interaction, the capturing agent builds a script that is suitable for executing multi-user tests:



Because EMPOWER/CS requires the multi-tasking features of UNIX to run multiple scripts, scripts are transferred to a UNIX driver machine where they are compiled and executed. With EMPOWER/CS, the UNIX driver can combine and execute several scripts simultaneously, and the scripts can be edited and enhanced to emulate more realistic client/server activity. Only the UNIX script driver and your SUT are required for the emulation.

When the script is executed on the UNIX script driver, it acts as a client to interact with the SUT emulating captured PC activities. As the script executes, the SUT assumes it is servicing SQL requests from an actual PC, thus, creating a true multi-user test environment:



The diagram illustrates the flow of data from a data warehouse to a data mart. On the left, a large data warehouse is shown with a stack of data blocks. Arrows labeled "SQL" point from the warehouse to two PCs in the middle, which are labeled "PC Events". From the PCs, arrows labeled "SQL" point to a data mart on the right, which is a smaller stack of data blocks.

The first step for load testing your client server environment with EMPOWER/CS is capturing PC and SUT interactions into a script file. When capturing scripts, changes to your client/server application are not required. The capturing tool only requires that you perform the most common user activities. To build complete scripts, EMPOWER/CS records mouse and keyboard strokes, SQL requests to the SUT, and data returned to the PC. The resulting scripts are transferred to the UNIX script driver.

EMPOWER/CS-V1.0.1

Then, to simulate the actual load on the client/server environment, you should combine and execute multiple scripts. You can duplicate individual scripts to simulate multiple users performing similar activities, and you can combine different scripts to emulate a complex set of user and application activity.

To produce the most accurate test results, you can edit your scripts to make them unique to your environment. Because EMPOWER/CS scripts are actually C language programs, they can be enhanced or supplemented as required. You can vary scripts by branching, altering data, and inserting loops. You also can change values entered in queries and updates during Capture (because in reality, all users would not query and update identical records continuously). User entries can be substituted by generating random values, sharing pools of input on the driver, accessing data returned from the SUT, or passing data between the scripts.

You can control the rate of transactions applied to the SUT while tests execute. Such controls as typing rate, thinking delays, constant and variable pacing, and synchronization points can be set at the start of a test and then adjusted as needed. Adjusting controls during the test permits load balancing and tuning.

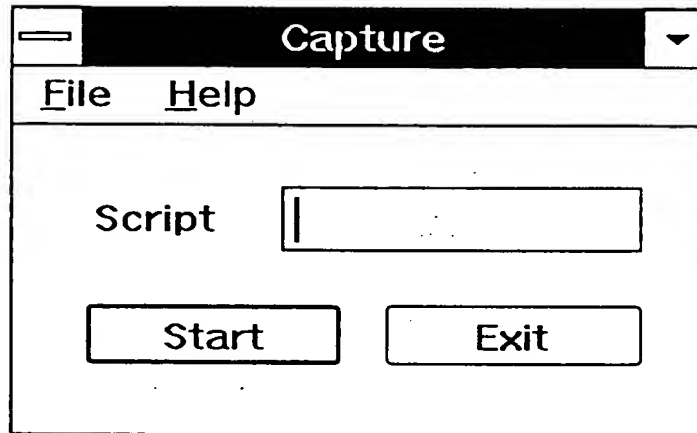
While tests are running, you can monitor scripts to verify progress, debug running scripts, and examine current response times. After your tests execute, you can generate reports that summarize interactive response time and peak throughput supported by your system. Valuable performance information such as response time averages, minimums, maximums, percentiles, transaction categories, and client processing times can be assembled within minutes.

2.4 EMPOWER/CS Tools

EMPOWER/CS includes and uses the following tools: Capture, Cscs, Mix, Extract, Report, Draw, Monitor, and Global Variables.

2.4.1 Capture

Capture is executed from Microsoft Windows on your PC to capture actual PC and SUT interactions in a script file. You simply activate the Capture icon, perform the user activity you wish to test, and then transfer your captured script file to the UNIX driver. Many options are available when capturing your script such as inserting user think times into the script file, adding comments and functions to the script, and automatically transferring scripts to the UNIX driver. The following example depicts the Capture command window.



A sample EMPOWER/CS script file, `script1.c`, follows. Such user and database activity as mouse events, logging on to the database, and processing a query were captured into this script (Some script content was left out for brevity):

(continued on following page...)

(continued on following page...)

```

Dbset(CUR1, FETCHSIZE, 64);
Fetch(CUR1);
while (GetNextRow(CUR1) != NOMOREROWS);

Endtimer(Accounts_Payable);

...

Begintimer("Accounts");

Commit(LOG1);

WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endtimer("Accounts");

Endscenario("script1");

```

2.4.2 C_{scc}

After the script file has been transferred to the UNIX driver, you should compile the script into a machine language version. The Csccl tool compiles the script which can be executed at the command line. Because the script is a C language program, additional C language statements can be added to enhance the script. In the following example, we will compile the script file `example1.c` into an executable binary:

```
$ cscd script1
```

The script now can be executed from the command line:

```
$ script1
```

In the following examples, suppose you have created four scripts. Your mix table, `table1`, could look like the following:

```
user1, query log1
user2, update log2
user3, accounts log3
user4, customer log4
```

With Mix, these scripts can be executed in a multi-user test:

```
$ mix
Mix: EMPOWER/CS V1.0.0, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95

mix> use table1
mix> start all
[user1] started
[user2] started
[user3] started
[user4] started
mix> user1 terminated (3/4) running
user2 terminated (2/4) running
user3 terminated (1/4) running
user4 terminated (0/4) running

mix> quit
$
```

2.4.4 Reports

Generating performance reports from one or more executed scripts is accomplished in two steps. First, you must use the Extract tool which parses each script's log file and records response time information in a set of flat ASCII files.

```
$ extract log1.1 log2.1 log3.1 log4.1
```

The Report tool then uses these ASCII files to produce statistical reports that describe response time and system throughput. (Note: You also may generate reports from your own statistical or graphics programs.) Report will generate an EMPOWER/CS standard report similar to the following:

```
$ report
```

EMPOWER Standard Report

Date: Fri Jan 27 14:49:14 1995
 Start time: 14:42:42
 Stop time: 14:43:23
 Duration: 00:00:41
 Mix: 4 users
 Unit: seconds

Scenario	Total	Finish	Thruput	Median	Average	Minimum	Maximum	Std-Dev
query	1	1	0.02	31.26	31.26	31.26	31.26	0.00
update	1	1	0.02	22.07	22.07	22.07	22.07	0.00
accounts	1	1	0.02	30.95	30.95	30.95	30.95	0.00
customer	1	1	0.02	22.18	22.18	22.18	22.18	0.00
Overall	4	4	0.10	26.56	26.62	22.07	31.26	4.49

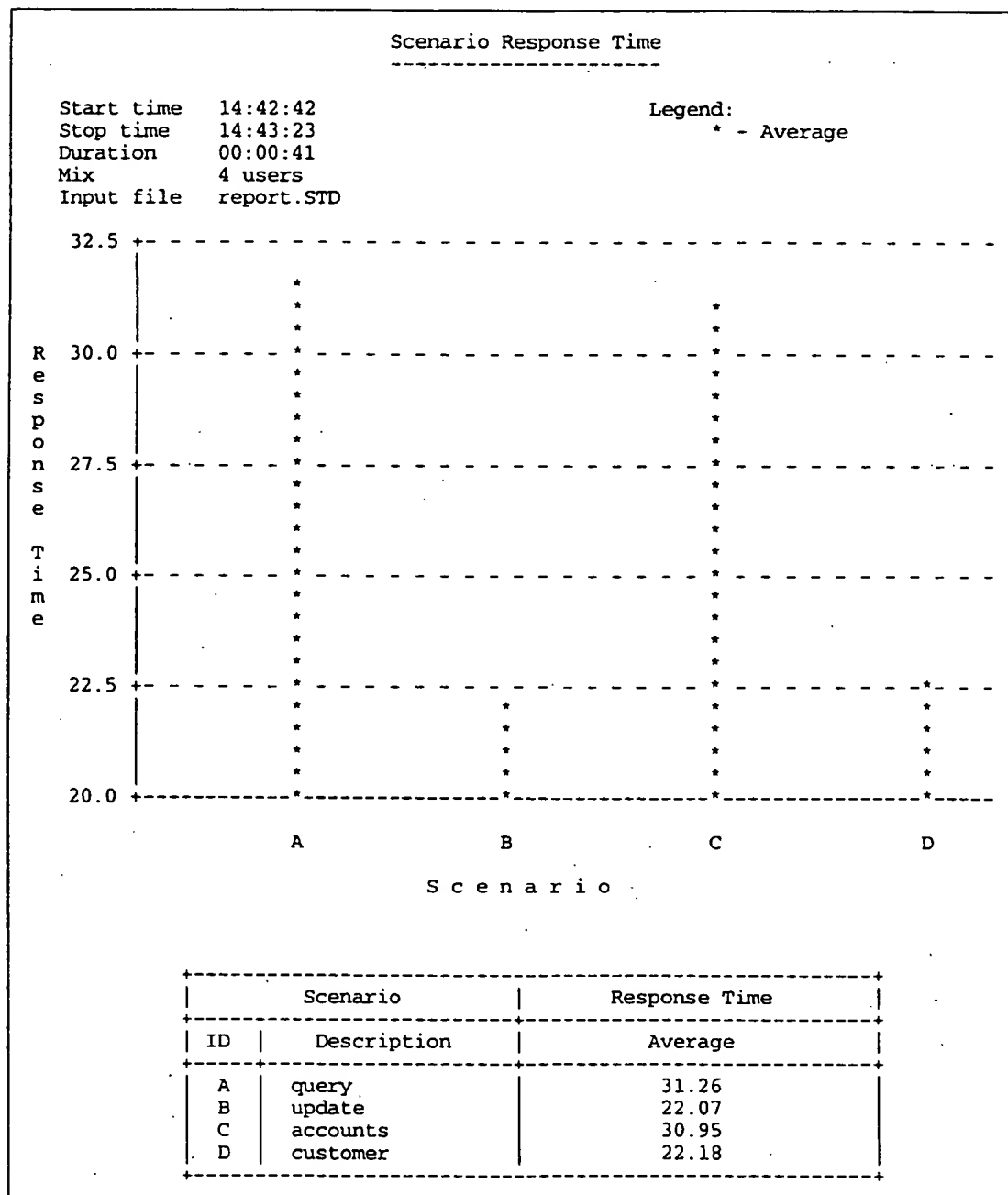
Function	Total	Finish	Thruput	Median	Average	Minimum	Maximum	Std-Dev
logout1	2	2	0.05	4.35	4.35	4.33	4.36	0.01

2.4.5 Draw

Draw accepts one or more EMPOWER/CS reports to produce bar charts that depict relationships among performance results. These relationships can summarize a

User's Guide—Script Development

single multi-user test or a series of multi-user tests in which each test contained a different user level or system configuration. The following example is a typical EMPOWER/CS bar chart:



2.4.6 Monitor

Monitor displays critical user information on your UNIX script driver during multi-user emulations. Information is listed in logically grouped views that can be sorted on any field in the view. With this tool, you can monitor response times, control script execution, track and correct script errors and timeouts, and view test progress. You also can identify and debug problem scripts, verify system load, and take a "snapshot" of user activity during a system bottleneck. Commands are provided to suspend, resume, or kill executing scripts during the multi-user test.

```
Fri Jan 27 14:42:58      EMPOWER/CS MONITOR V1.0      (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 4      Running      ScriptId Sorting ^ Interval 5

ScriptId  __Script State      LastEvent      CurrentWindow
user001   query  bpush  ><ButtonPush><ButtonPush><ButtonPush>      Employee Records
user002   update  appwt  ButtonPress>c:\acct\acct^M<ButtonPush>      Acct Application
user003   accounts  appwt  nPress><LeftButtonPress>c:\acct\acct^M      Accounting
user004   customer  type  <LeftButtonPress><LeftButtonPress>c:\      Run
```

2.4.7 Global Variables

The Global Variables tool, EMPOWER/GV, provides advanced control over multi-user emulations by creating and controlling global variables that are shared among executing scripts. Global variables can be defined to direct scripts to terminate gracefully when they run an indefinite process. They also may be used to synchronize the execution of multiple scripts. Elements of EMPOWER/GV include various commands and functions that control access to variables; read, update, or test values of variables; and control shared memory segments.

3.0 Installation

EMPOWER/CS was designed to operate in an environment in which the SUT can not distinguish between actual PC users and the UNIX script driver. The PC, UNIX script driver, and SUT must be connected along the same communication network. The EMPOWER/CS software does not affect communication between the user PC and the SUT as it captures their activity.

Before a Capture session begins, the PC will run a licensing check with the UNIX driver to verify execution of Capture. If the PC can not communicate with the UNIX machine, you will not be able to execute Capture. The PC must also be linked to the UNIX script driver so that it may drive the PC during script execution in Display mode.

If the PC can not communicate with both the UNIX driver and the SUT, then EMPOWER/CS can not capture client/server activity. If the UNIX driver and the SUT can not communicate, then the UNIX driver can not execute a multi-user emulation.

The Capture, Display, and Tools elements of EMPOWER/CS run on a Microsoft Windows PC and the remaining EMPOWER/CS software runs on the UNIX script driver. So that the PC can communicate with the UNIX script driver for the licensing check and for script execution in Display mode, the PC must have installed the dynamic link library winsock.dll Version 1.1. The PC application communicates with the database according to how you have set up your client/server environment. Both the PC and the UNIX driver must have the appropriate database libraries installed to communicate with the database on the server.

When a script is executed, the PC no longer is required because the UNIX driver replaces the PC user to interact with the SUT. However, if you choose to observe a script as it progresses, you can activate the "Display" option for the script to display captured user activity on the PC.

3.1 The EMPOWER/CS Environment

The following equipment is required to capture user activity with EMPOWER/CS:

- ❑ One IBM-compatible PC with keyboard, mouse, and monitor running Microsoft Windows software, winsock.dll Version 1.1, database libraries that assist with communicating with the SUT, and the EMPOWER/CS components Capture, Display, and Tools
- ❑ The UNIX script driver machine with database libraries that provide comparable communication with the SUT and EMPOWER/CS software
- ❑ The database server
- ❑ One communications network connecting the PC, the UNIX script driver, and the database server

3.2 Installing the User PC, the UNIX Script Driver, and the Database Server

You should follow the appropriate installation procedures described in the owner's manuals for the user PC, the UNIX script driver, and the database server.

3.3 Installing the EMPOWER/CS Software

You must complete two installations to fully operate EMPOWER/CS. The EMPOWER/CS components (Capture, Display, and Tools) must be installed on the PC and the remaining EMPOWER/CS software must be installed on the UNIX script driver.

3.3.1 Installing EMPOWER/CS on the UNIX Script Driver

The EMPOWER/CS software package is installed into four directories. The `bin` directory contains binary programs, the `lib` directory contains libraries of compiled functions, the `h` directory contains header files, and the `Install` directory contains installation files.

You will need at least 15 megabytes of free disk space on your UNIX script driver to install and run EMPOWER/CS. You also will need additional space to store scripts and log files. The UNIX script driver must run a version of the UNIX operating system, the appropriate database libraries, and a C language compiler.

To install the software, follow these steps:

Step 1: Create a directory for the EMPOWER/CS software.

The simplest directory to create is `/usr/empower`. Some customer sites create a separate EMPOWER/CS account and install the software in the home directory of that account. If you create an EMPOWER/CS account, we suggest you call it `empower` in the event PERFORMIX must access the account for future maintenance or upgrades.

The remaining instructions are based on installing EMPOWER/CS in `/usr/empower`.

Step 2: Log in as the user who will own the EMPOWER/CS software.
(Note: This user typically is `empower`.)

Step 3: Change to the directory where the software will be stored:

```
$ cd /usr/empower
```

Step 4: Insert the EMPOWER/CS distribution tape into a tape drive. You must either use the default tape drive or determine the `/dev` name of the tape drive you will use.

Step 5: Use the `tar` command to read in the tape. Some examples of `tar` commands are listed below:

```
$ tar xov
$ tar xovf /dev/rSA/qtape1
```

Step 6: Change to the `Install` directory and execute the `emininstall` command. You will be instructed to contact PERFORMIX and provide your machine identification code to obtain your installation password, for example:

```
$ cd Install
$ ./emininstall
```

You will receive a message similar to the following:

Please print the `../Install/machineid` file, note your name, fax number, and phone number, and fax it to Performix at 703-893-1939.

We'll generate your installation password and fax it back as soon as possible. When you get your installation password, re-run `emininstall`.

If you are unable to send a fax, please call Performix at 703-448-6606.

This message is stored in the `machineid` file which can be printed and faxed to PERFORMIX. You will be given a special password consisting of all alphabetical characters, for example, JKLMNOPQRS-ABCDEFGHIJLMNOPWXYZ.

```
$ ./emininstall
Installation password:  JKLMNOPQRS-ABCDEFGHLMNOPWXYZ
installing ../lib/empowercs.a...
installing ../lib/empowercsm.a...
installing ../bin/csccl...
installing ../bin/draw...
installing ../bin/extract...
installing ../bin/gv_cmds...
installing ../bin/mix...
installing ../bin/mon...
installing ../bin/report...
```

```
$ ranlib lib/*.a
```

3.3.2 Configuring the UNIX Driver for EMPOWER/CS

Step 2: Define your shell environment variable `EMPOWER` to be the directory in which `EMPOWER/CS` was installed.

For example, if you are a C-shell user, add the following line to the `.cshrc` file in your home directory:

```
setenv EMPOWER /usr/empower
```

If you are a Bourne-shell user, add the following line to the `.profile` file in your home directory:

```
EMPOWER=/usr/empower; export EMPOWER
```

Step 3: Add `$EMPOWER/bin` to your path. This step is accomplished by modifying the `PATH` statement in the `.cshrc` or `.profile` file in your home directory. You must modify your `PATH` to be below the setting of the `EMPOWER` environment variable.

If you are a C-shell user, the `PATH` setting statement in your `.cshrc` file should be:

```
set path = ( . /bin /usr/bin $EMPOWER/bin )
```

If you are a Bourne-shell user, the `PATH` setting statement in your `.profile` file should be:

```
PATH=./bin:/usr/bin:$EMPOWER/bin
```

Step 4: Log out of your system and then log back in. This step causes your environment variables and `PATH` to be set correctly.

3.3.3 Installing EMPOWER/CS on the PC

When you install the EMPOWER/CS components onto your PC, EMPOWER/CS libraries and directories are created automatically. By default, the EMPOWER/CS libraries are placed under the directory `empower` which also includes a `bin`

directory of binary programs. If you wish to place these files under a directory other than `empower`, specify a new directory during installation when prompted.

You will need at least five megabytes of free disk space on your PC to install and run EMPOWER/CS. You also will need additional space to store scripts and log files. The PC must include the libraries of the database server, the DOS operating system, and Version 3.1 of Microsoft Windows.

To install the EMPOWER/CS components onto your PC, follow these steps:

- Step 1:* Insert the Installation diskette into your disk drive.
- Step 2:* From the DOS prompt, switch to the appropriate disk drive. Execute the DOS install command from the correct disk drive by typing the name of the drive and `csinstal` (Example: `a:\csinstal`). If Microsoft Windows is already running, in the Program Manager window, choose Run from the File Menu. In this window, type the name of the drive and the install command (Example: `a:\csinstal`) and select OK.
- Step 3:* Follow the instructions that appear on screen for the remainder of the installation.

3.4 Before Starting Capture

Before you can Capture with the EMPOWER/CS software, a licensing verification must occur between the PC and the UNIX script driver. This licensing check will occur when you start Capture.

Before starting Capture, you must run `eld` on the UNIX machine. You should start `eld` as super-user (`root`) and you must set and export the EMPOWER environment variable to the local directory that contains the EMPOWER/CS software.

```
# EMPOWER=/usr/empower
# export EMPOWER
# eld
```

You now should be able to use the EMPOWER/CS software. Confirm the installation by starting Microsoft Windows on your PC and activating the EMPOWER/CS window and then the Capture icon.

The software installation and setup are now complete. Remove the EMPOWER/CS tape from the tape drive and store it, with the EMPOWER/CS diskette, in a secure location.

The EMPOWER/CS Capture tool is used to capture communications between a user PC and the SUT to build executable scripts. After you complete a Capture session, the assembled scripts can be executed from a UNIX script driver to emulate numerous users of the client/server system.

When activated, Capture records all interactions between the PC and SUT including mouse and keyboard strokes, SQL requests to the SUT, and data transmitted to the PC. Automatically, this information is stored in a script file with a ".c" extension in a specified directory on your PC to be used for subsequent script execution. The .c extension implies that the file is recorded in C language syntax for later C compilation and execution. Refer to Sections 5 and 6 of this manual for more information on script compilation and execution.

The following script segment illustrates the types of user functions and database traffic contained in a script. This script contains such captured activity as user entries, connection and logging on to the database, and execution of a query:

(continued on following page...)

[illegible]

For a more detailed explanation of script content, please refer to Section 7 Script Content and Enhancement of this manual.

4.1 Suggestions for Executing Capture

If you plan to execute EMPOWER/CS scripts in Display mode, you must ensure that the steps captured in the application to be tested are repeatable, meaning that all captured mouse activity can be *precisely* repeated by the UNIX script driver machine during script execution. Capturing mouse clicks in each activated window is not recommended because the recorded mouse locations, i.e., xy coordinates on screen, may not apply during script execution. For instance, windows do not always open in the same location each time they are activated. During script execution in Display mode, the recorded xy coordinates in the mouse event functions could be invalid and cause the script to malfunction by not activating the correct locations to activate applications or commands. Maximizing windows will ensure that all mouse actions are repeatable because the window will cover the entire screen allowing recorded xy coordinates to remain the same in Display mode.

Since keyboard presses and commands are more precise, we recommend that you use the keyboard as much as possible to activate windows, commands, or applications. Some keyboard capabilities are listed below:

- Alt + Space Bar to open system menus
- Alt + F to open file menus
- Control + F4 to close program windows
- Arrow keys to move through menus or among icons

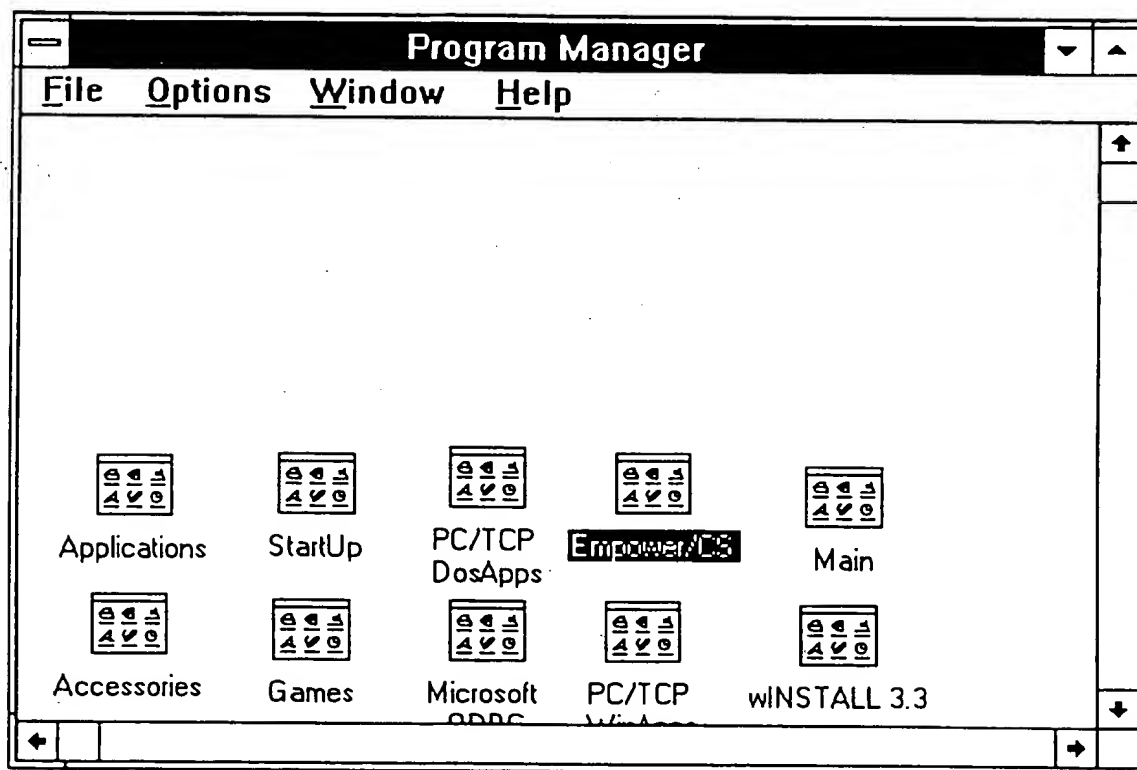
Refer to your MS Windows user guide for full instructions on using the keyboard to control your Windows desktop.

Capture is executed from Microsoft Windows on your PC.

If Windows is not currently running on your PC, from the DOS prompt, type "win" and press return.

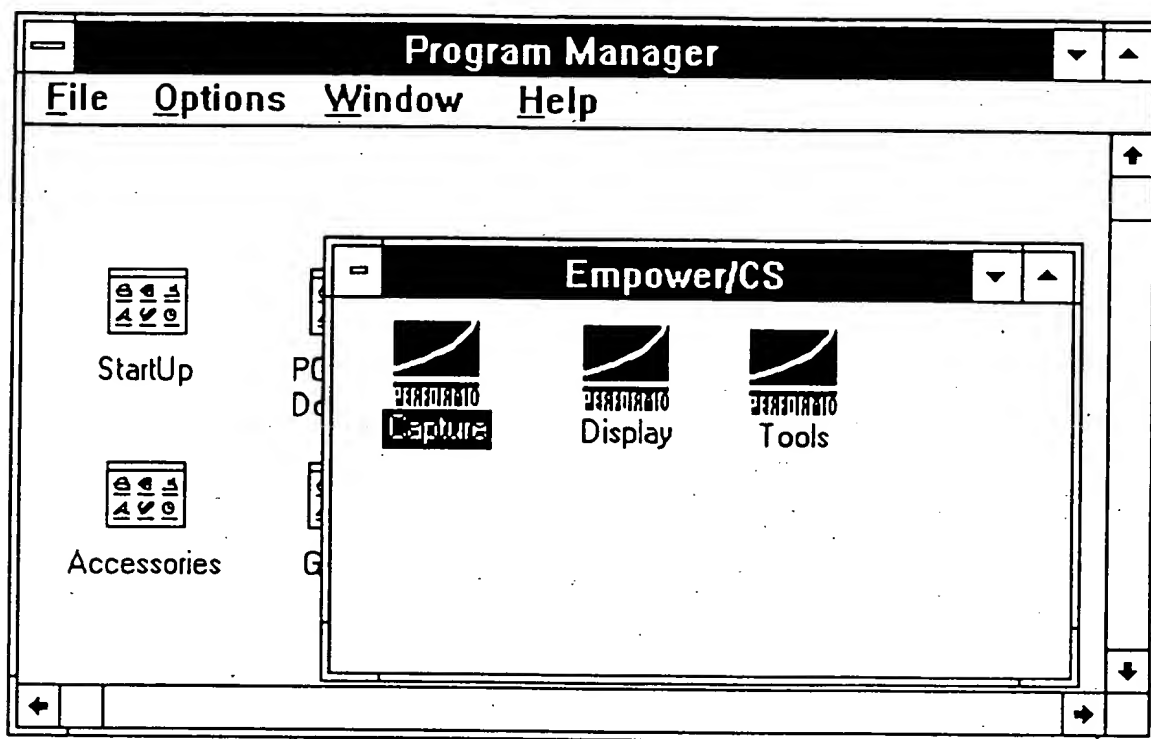
The Windows desktop and Program Manager window will open containing various program group icons. Activate the **EMPOWER/CS** program group.

Example:



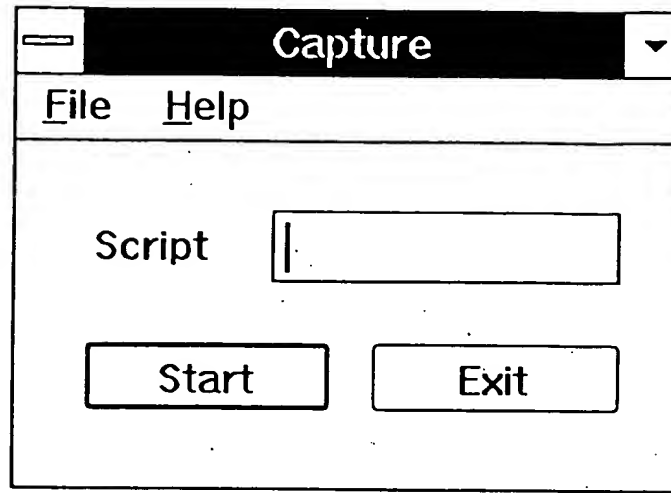
The following EMPOWER/CS window will open which includes three program-item icons: Capture, Display, and Tools.

Select the Capture icon:



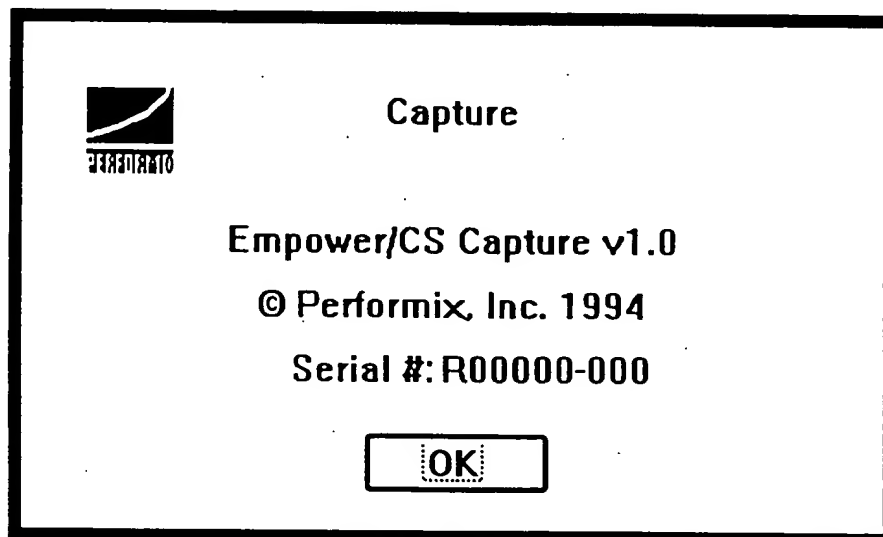
Before you may begin Capture, the PC must set up an initial communication or licensing check with the UNIX script driver. If this initial contact can not be made, you will not be able to execute Capture and will receive an error. *Note:* You must be sure to run `elid` as root on the UNIX machine before starting Capture on the PC. (Refer to Section 3.3.3 of this manual).

If the licensing check was successful, the following Capture command window will open prompting you to enter a script name:



This Capture window includes two items in the Menu Bar: **F**ile and **H**elp. The File menu includes **O**ptions..., **D**irectory..., and **E**xit. The Help Menu (located in all the EMPOWER/CS windows) includes **A**bout... for listing EMPOWER/CS copyright information.

The About Capture screen is shown below:



Before you enter a script name to begin capturing, you may wish to change some of the Capture menu options. Under the **File Menu**, select **Options...**

4.2.1 Options

When you choose Options..., the following Options window will open:

Capture Options.

Think Threshold ☐ Database traffic only

Bring to front with ☐ Insert timers

View script ☐

Transfer script after capture ☒

Host

Username

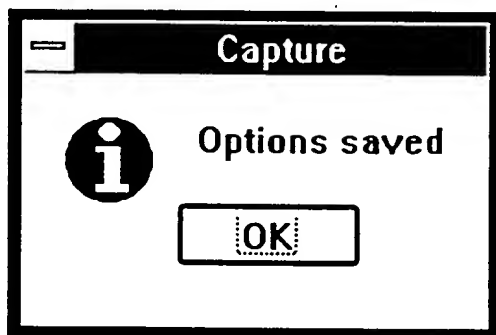
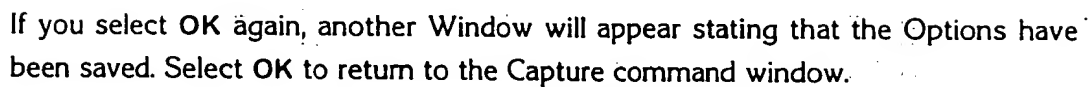
Password

Remote Directory

License Daemon

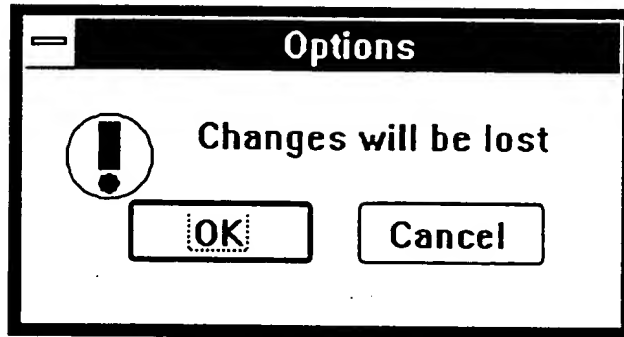
These Options are described more fully in subsequent sections.

Example:



To ignore any changes you made, select **Cancel**. A verification window will appear stating that the changes will be lost. Select **OK** to return to the Capture command window.

Example:



In the **Options** window, you may change any of the Capture options during your Capture session, except **Database traffic only** and the **Database... Chooser**.

4.2.1.1 Think Threshold

This option specifies the number of seconds that EMPOWER/CS will wait before inserting a `Think()` function into the script .c file. (Refer to Section 7 Script Content and Enhancement for a more detailed explanation of think time functions.) For example, suppose you specify a Think threshold of 2 seconds. If no activity is captured after two seconds, EMPOWER/CS will insert a think time function of at least two seconds and the time elapsed until the next action occurs.

You may specify any two digit number of seconds for this option.

The following example demonstrates a View script window during Capture:



Database traffic includes such functions as `Open()`, `Parse()`, `()`, `Exec()`, `Fetch()`, etc. which are inserted into the script when the client application interacts with the database. No user interactions, such as mouse button presses or `Type()` functions, are inserted into the script file. However, the amount of time taken to type characters or move the mouse will be included in the `Think()` times that are recorded into the script. (Refer to Section 7 Script Content and Enhancement for a full description of these script functions.)

4.2.1.5 Insert Timer

If you specify this option, `BeginTimer()` and `EndTimer()` functions are inserted automatically into the script during Capture to mark database traffic. These functions will be inserted around database traffic that occurs between two user events. A user event such as pressing the Return key on the keyboard or

The following example demonstrates `BeginTimer()` and `EndTimer()` captured into a script file around a database query. Notice the user event captured as the functions' parameters was a `ButtonPush()` event, when a user pressed the button "Payable":

```
CurrentWindow("Accounts",147,66,533,360);
ButtonPush("Payable",267,213);

AppWait(0.05);
WindowRcv("SfPtCwCwCwCwCwCwCwCwCwCwCwCw");

Begintimer("Accounts_Payable");

Dbset(CUR1,DEFER,TRUE);

Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD,
ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

...

Endtimer("Accounts_Payable");
```

4.2.1.6 Database Chooser

This option allows you to choose the database on the server that you will access during the Capture session.

Database

- ✓ Oracle
- Sybase

OK Cancel

Once a script is captured, it must be transferred to the UNIX machine to be executed. If you select this option, your script .c file will be transferred automatically when you stop Capture.

EMPOWER/CS-V1.0.1

Example:

Transfer script after capture ☒

Host

Username

Password

Remote Directory

License Daemon

OK Cancel

In the Options window, enter the **Host** name (the name of the UNIX driver), the **Username**, the **Password** (the user's password, if any), and the **Remote Directory** (the path on the UNIX driver where you wish to transfer the script).

If large amounts of data (i.e., images, large text files, etc.) are input to the database during Capture, such data is inserted into separate data files. These data files will be transferred to the UNIX driver machine with the script if the **Transfer script after capture** option is selected. The data files are associated to the script with a **.d** extension and a number that is incremented for each file. For example, two data files created for the script, `script1.c`, would be named as follows:

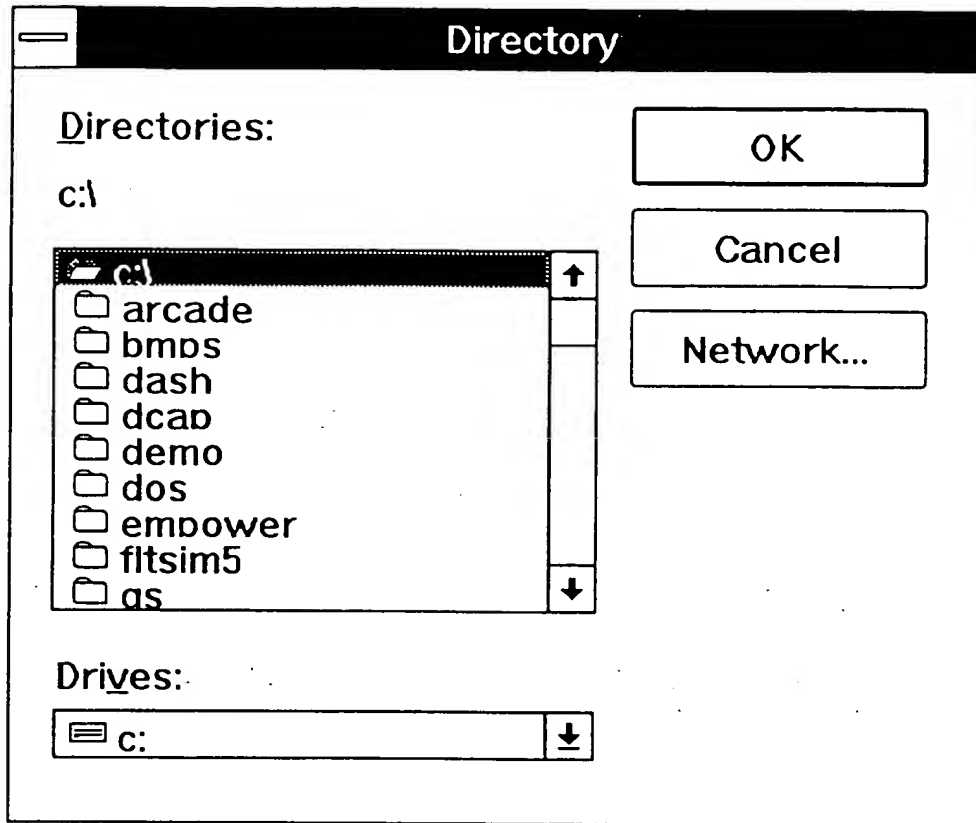
```
script1.d01
script1.d02
```

If you prefer, you may transfer the script file manually with the Transfer tool under EMPOWER/CS Tools. Refer to Section 8 EMPOWER/CS Tools for more information on manual script transfer.

If you specified a license daemon during EMPOWER/CS installation, this option lists the name of that license daemon. If you need to change or specify a license daemon machine (which is the UNIX script driver used for your emulation), you must enter the new name in this dialog box.

You may use this File Menu option to store scripts in a different directory on your PC.

Select **Directory...** under the **File** Menu. The following window will open designating the directory and disk drive where your scripts currently are being saved.

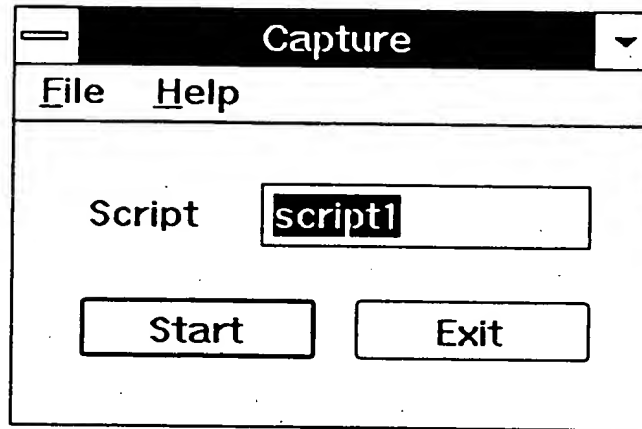


Select from the Directory list the name of the directory or the disk drive to which you wish to change, and then select **OK** or **Cancel** as appropriate.

4.2.3 Capturing Application Activity into a Script File

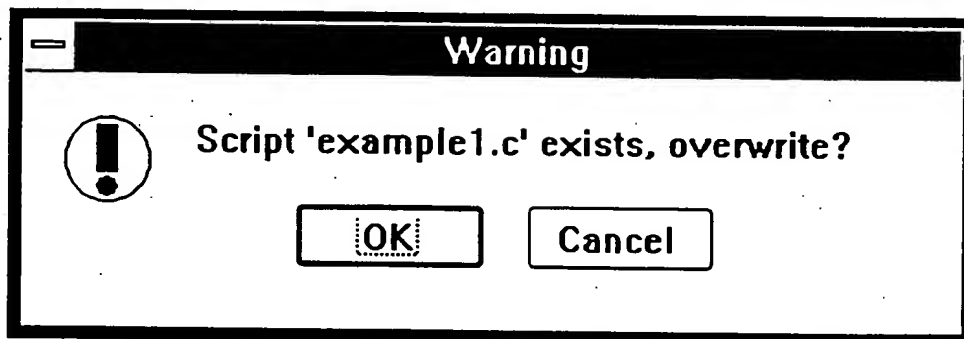
In the Capture command window, enter the script name you are capturing. The script file name can include up to eight alphanumeric and underscore characters. If you enter more than eight characters or other character types, Capture will ignore them. During Capture, the script file is stored automatically in the specified directory in an ASCII file with a ".c" extension.

Example:



When you have entered a script name, select **Start**.

If the script name you enter already exists, the following warning message will appear, asking if you wish to overwrite the existing file. Select **OK** or **Cancel** as appropriate.



Upon selecting **Start**, the Capture window will minimize, or iconify, into the lower left corner of the desktop signifying that all subsequent user and database actions are being captured.

Now, you should open the application to be tested. Refer to your application's user guide for full operating instructions. Remember to maximize your application window upon activation if possible and use keyboard commands instead of the mouse during your Capture session.

Please note that user activities are captured only from Windows standard applications. For example, EMPOWER/CS does not capture user activity from DOS applications running under Windows.

At this time, you should perform the application activity you wish to test.

4.2.4 Comments, Functions, and Timers

You can add timers functions, or comments, to your script file while you are capturing by activating the Capture icon. The following Capture command window will open. When the Capture window appears on screen, all capturing activity halts temporarily.

The screenshot shows a window titled "Capture" with a menu bar containing "File" and "Help". Inside the window, there is a container with three rows of input fields and buttons:

- Timer:** A text input field followed by a "Begin" button.
- Function:** A text input field followed by a "Begin" button.
- Comment:** A text input field followed by an "Insert" button.

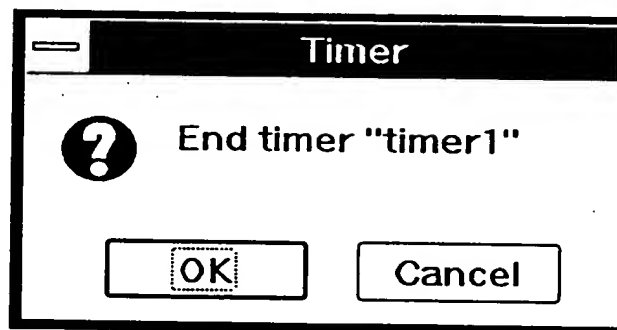
At the bottom of the window, there are two buttons: "Resume" and "Stop".

Inserting timers allows you to mark specific activity for response time measurement. C language functions most commonly are used to help define a script segment for looping purposes. They also are used to define a common interaction, such as logging onto or off of the SUT. You should add comments to your script file to add context to the script for editing purposes.

4.2.4.1 Inserting Timers

You can manually insert the functions `BeginTimer()` and `EndTimer()` into your script to mark activity for response time measurement. You must enter the name of the timer in the **Timer** dialog box of the **Capture** window and select **Begin**. Select the **Resume** button to return to the **Capture** state to capture the activity you wish to measure.

When you have captured all needed activity, activate the **Capture** icon to open the **Capture** window. Select the **End** button next to the **Timer** dialog box. The following window will appear to verify the end of the timer:



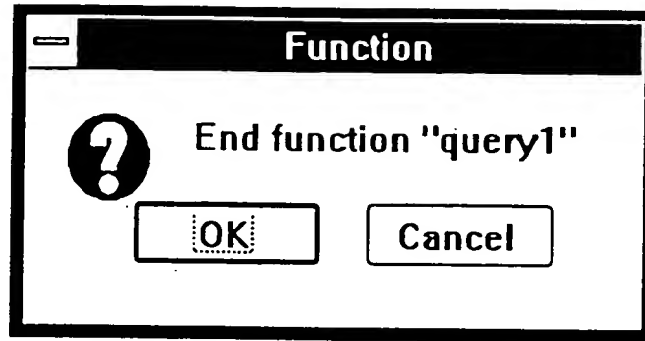
If you have finished inserting the timer, choose **OK** in this verification window. Your desktop then will return to the **Capture** state.

4.2.4.2 Inserting Functions

To create a C function, enter the name of the function in the **Function** dialog box and select **Begin**. Select the **Resume** button to return to the **Capture** state to capture the function activity.

After you have completed capturing the function, activate the **Capture** window and select the **End** pushbutton next to the **Function** dialog box.

The following window will appear verifying the end of the function:



If you have finished inserting the function, choose **OK** in this verification window. Your desktop then will return to the Capture state.

When you create a function, a function call is inserted within the script and the actual function (that includes captured activity) is placed at the end of the script file. Automatically, the EMPOWER/CS functions `Beginfunction()` and `Endfunction()` are inserted around the captured function. After a script is executed, `Beginfunction()` and `Endfunction()` cause time stamps to be recorded in a log file which is used to create detailed response time reports. Marking functions in such a way allows you to measure response time for specific activities in your application. Refer to Section 6.4 The Log File in this manual and to Section 3.3 of the *Multi-User Testing* manual for more information on time stamps.

The following example demonstrates a function `logout()` that was captured into the script file, `script1.c`:

```
logout()  
{  
  BeginSource("script1.c");  
  BeginFunction("logout");  
  
  Think(4.66);  
  
  LeftButtonPress(202,99);  
  
  LeftButtonPress(236,244);  
  
  AppWait(0.28);  
  WindowRcv("ScDwAc");  
  
  CurrentWindow("Capture - script1",21,690,57,726);  
  
  Commit(LOG1);  
  
  WindowRcv("Pt");  
  Logoff(LOG1);  
  CloseEnv(ORACLE);  
  
  EndFunction("logout");  
  EndSource();  
}
```

`Beginsource()` and `Endsource()` statements are automatically inserted around `Beginfunction()` and `Endfunction()` to prepare the function as a source file. For instance, during a multi-user emulation, you may wish to break the function out of the script into its own separate file that could be called by multiple scripts performing the same function. Modular script design is achieved by storing one or more functions in separate script source files. The C language `cc` compiler allows functions stored in these files to be compiled separately and then linked to the primary script file. `Beginsource()` and `Endsource()` specify the source file used during script execution.

EMPOWER/CS organizes functions as described in this section to format the script as a C language program. Refer to Section 5.2.3 Modular Script Design for a more detailed description of C function formatting.

4.2.4.3 Comments

You may insert a C comment into your script by entering the comment in the **Comment** dialog box of the **Capture** window and selecting **Insert**.

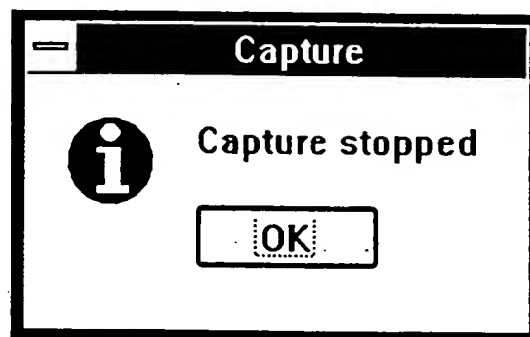
Select the **Resume** or **Stop** pushbutton as appropriate when you have entered your comment. If you select **Resume**, your desktop will return to the **Capture** state.


4.2.5 Completing Your Capture Session

After you complete the application activity to be tested, close the application.

Activate the Capture icon to halt the Capture session. The Capture window will appear.

If you are ready to end the Capture session, select the **Stop.** button or **Exit** from the **File** menu. The following window will appear verifying that Capture has stopped:



A screenshot of a Windows-style warning dialog box. The title bar is black with the word "Warning" in white. The main area has a white background. On the left is a black circle containing a white lowercase "i". To the right of this icon, the text "No database traffic captured" is displayed in a bold, black, sans-serif font. At the bottom center is a button with a black border and the text "OK" in a bold, black, sans-serif font.

Warning

i No database traffic captured

OK

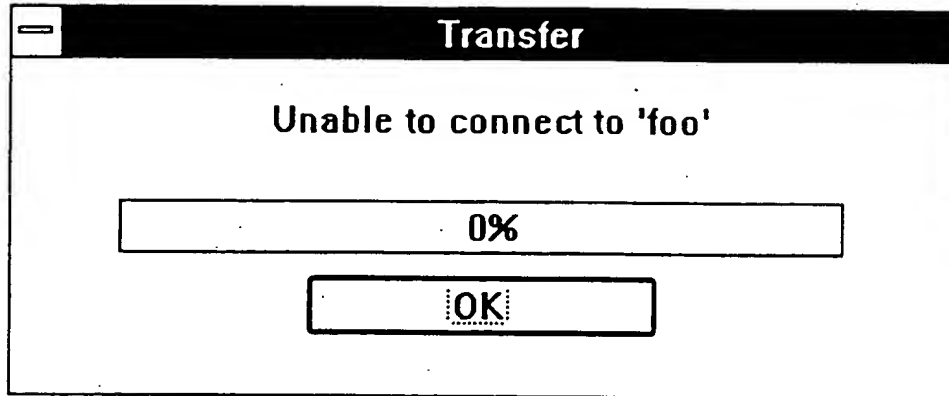
Transfer

Attempting to connect to 'indy'

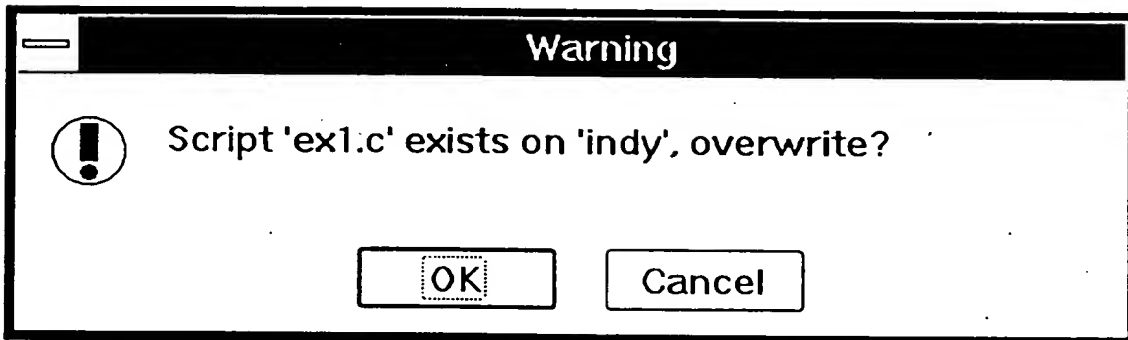
0%

Cancel

If the PC is unable to connect to the UNIX driver, a message window similar to the following will appear:



If the script .c file already exists on the UNIX script driver upon a successful transfer, the following window will come up asking you to overwrite the existing script .c file. Choose OK or Cancel as appropriate:



If you did not select the automatic transfer option, you may transfer the script manually using the EMPOWER/CS Transfer Tool (see Section 8.0 EMPOWER/CS Tools) or a third party File Transfer Protocol (FTP) software.

The Capture session is now complete. At this point, you can begin to capture another script or you can close Capture by selecting "Exit" from the File Menu.

If you choose **Exit**, you now are ready to compile your script(s) with **Cscc** on the UNIX script driver. **Cscc** is the EMPOWER/CS tool that compiles your scripts into machine language binaries for execution. If you wish to edit your script .c file, you should do so with a system editor such as **vi** after the script file has been transferred to the UNIX script driver. Refer to Section 7.0 Script Content and Enhancement for information on editing your scripts.

009207 46626960

EMPOWER/CS-V1.0.1

```
$ csc -
```

EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Usage:

```
csc [-EFOacghsvm] [-o ofile] script ...
```

Options:

-E	Preprocesses the script and writes C to stdout
-F	Inserts function declaration (implies -c option)
-O	Optimize script binary
-a afile	Create an archive (.a file) from the named files
-c	Compile but do not link (creates .o file)
-g	Includes symbol table in script binary for debuggers
-h	Excludes help information from the script binary
-s	Prevents strip of script binary
-v	Print verbose cc command line
-m	Excludes Monitor code from script binary
-o ofile	Name output binary

Notes:

Before running CSCC, set the environment variable EMPOWER to the directory that contains the EMPOWER software.

Set the environment variable E_CFLAGS to any additional cc compile options you want.

If necessary, this environment variable can be set to more than one database. However, you should only specify the database(s) you used during the Capture session.

Example:

```
$ setenv E_DATABASES ORACLE7
```

With each E_DATABASES environment variable you must set the Oracle or Sybase environment variables.

If compiling with an Oracle database, set the environment variable ORACLE_HOME to the path where Oracle resides on the server.

Example:

```
$ setenv ORACLE_HOME /usr/oracle
```

If compiling with a Sybase database, set the SYBASE environment variable to the appropriate path.

Example:

```
$ setenv SYBASE /usr/sybase
```

You also can pass flags when invoking the C compiler by defining an environment variable E_CFLAGS and setting the variable equal to your compilation flags. The flags will be inserted after the cc command and before the file name.

If you require additional libraries during the load phase of the C compilation, define the E_LIBS environment variable equal to the names of the libraries.

Example:

```
$ setenv E_LIBS "-la"
```

5.3 Compiling with Csc

To compile a script, you must enter the `csc` command with the name of the source script. (Note: You do not need to specify ".c" in the script file name because Csc automatically searches for a file with this extension.)

Example:

```
$ csc example1
Csc: EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
cc -s -o example1 ./csc7086.c /usr/local/lib/*.a
```

Csc will look for the specified script file in the current directory. If you have saved the script file in a different directory, you can specify the path of the directory as part of the file name specification.

```
$ csc /usr/empower/scripts/example1
```

During script compilation with Csc, the source script is converted and placed in a temporary file in the current directory. Then, the C compiler is invoked to compile this temporary file. During the load phase of compilation, a special library of EMPOWER/CS functions is included. Once the compilation is complete and the executable file has been created, the temporary file is removed automatically and the executable file remains in the current directory.

Csc creates an executable file called `example1` (with no extension) which can be executed by typing this file name at the command line.

5.3.1 Specifying the Binary Name with -o

By default, Csc creates a binary version of the script with the same name as the script file but without the .c extension. If you want to save the binary with a

```
$ csc -o database1 query.c
```



```
LeftButtonPress(457,617);

WindowRcv("DwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");
CurrentWindow("Run",429,491,880,764);
Type(:c:\\acct\\acct^M");

function1();

...

CurrentWindow("Capture - script1",21,690,57,726,"Min");
Commit(LOG1)

Close(CUR1);

WindowRcv("Pt");
Logoff(LOG1);
Closeenv(ORACLE);

Endscenario("script1");
)

function1()
(
    Beginsource("script1");
    Beginfunction("function1");

    AppWait(0.22);
    WindowRcv("CwCwCwCwCwCwCwCw");
    Openenv(ORACLE,VERSIONV6V7);

    Username(LOG1, "scott/tiger@SUT");
    Password(LOG1, "");
    Logon(ORACLE, LOG1);

    Open(LOG1,CUR1);

    WindowRcv("AcSf");
    CurrentWindow("Accounting Application",189,82,685,449);

    Endfunction("function1");
    Endsource();
)
```

The function syntax can be inserted manually or may result from inserting a function during Capture. (See Section 4.2.4.2 Inserting Functions.)

Your script also can access functions that are stored in other script files. Modular script design is achieved by storing one or more functions in separate script source files. The C language `cc` compiler allows functions stored in these files to be compiled separately and then linked.

You must use the `-c` option of the `csc` command to create object files (with an ".o" extension) for each compiled script.

Examples:

```
$ csc -c func1
$ csc -c func2
$ csc -c func3
```

For a script to access these functions, a function call must be inserted within the script and the object files must be linked with the source script. This is accomplished by compiling the script source file and listing the object files as parameters of the `csc` command, as shown below:

```
$ csc script1 func1.o func2.o func3.o
```

In the above examples, three separate functions are stored in the script source files `func1.c`, `func2.c`, and `func3.c`. Each is compiled separately and then linked to the source script file, `script1.c`.

Individual object files may be stored and used by other scripts or included in an EMPOWER/CS library.

When the `-c` option of the `csc` command is used, the function stored in a separate file must include standard C language function formatting (e.g., braces, functions, etc.).

If a script is compiled from multiple source files, each source file must be specified as a source file by including `Beginsource()` and `Endsource()` statements. When editing your script, you should insert the `Beginsource()` function at the entry point of the source file, typically just before the first executable statement in each function. You should insert the `Endsource()` function at the exit point of the file, typically just after the last executable statement in each function.

The following example script segment demonstrates a typical source file:

```
logon1()  
{  
  Beginsource("logon1");  
  
  AppWait(0.17);  
  WindowRcv("SfPtCwCwCwCwCwCw");  
  Openenv(ORACLE,VERSIONV6V7);  
  
  Username(LOG1, "scott/tiger@SUT");  
  Password(LOG1, "");  
  Logon(ORACLE, LOG1);  
  
  WindowRcv("SfAcSfDwPtPtPtPtPtPt");  
  
  CurrentWindow("Accounting",413,679,1029,771);  
  
  Open(LOG1,CUR1);  
  
  WindowRcv("AcSf");  
  
  CurrentWindow("Accounting Application",189,82,685,449);  
  
  Endsource();  
}
```

Note: If you define C functions during your Capture session, `Beginsource()` and `Endsource()` are placed around `Beginfunction()` and `Endfunction()` statements in the event you wish to break the function out later into a separate source file.

5.3.4 Extending Modular Script Compilation with -F

The `-F` option of the `csc` command extends the capabilities of modular script compilation to handle functions that do not include standard C function formatting. This option places the required formatting into the script automatically.

For example, elements of the `logon1()` function may be contained in the following script file called `logon1.c`:

```
Beginsource("logon1");

AppWait(0.17);
WindowRcv("SfPtCwCwCwCwCwCw");
Openenv(ORACLE,VERSIONV6V7);

Username(LOG1, "scott/tiger@SUT");
Password(LOG1, "");
Logon(ORACLE, LOG1);

WindowRcv("SfAcSfDwPtPtPtPtPtPt");

CurrentWindow("Accounting",413,679,1029,771);

Open(LOG1,CUR1);

Endsource();
```

To compile this file into the object file `logon1.o`, use the `-F` option:

```
$ csc -F logon1
```

Csc adds the function definition logically to the source file and compiles it into the object file `logon1.o`. The `-F` option may not be used if the function is required to accept arguments.

5.3.5 Automatic Creation of Function Archives

EMPOWER/CS can create archive and function libraries automatically with the `-a` option of `csc`. This option is used as follows:

```
$ csc -a logon1 func1 func2
```

The `-a` option is followed by a list of script file names to be included in the archive. If a `.o` file exists for the given script file, the `.o` file also will be added to the archive. If a `.o` file does not exist, Csc will search for a `.c` file with the provided name. If a `.c` file is found, it automatically will be compiled with the `-c` option, and the resulting `.o` file will be included in the archive.

The name of the archive will be the first script name given, with a `.a` extension. The above example would create an archive file called `logon1.a`, which would include the files `login.o`, `func1.o`, and `func2.o`.

The `E_LIBS` environment variable must then be set to include the archive file. Future execution of Csc will include the new archive. Set the `E_LIBS` environment variable as follows:

For Bourne shell users:

```
$ E_LIBS=logon1.a; export E_LIBS
```

For C Shell users:

```
$ setenv E_LIBS logon1.a
```

When the script is compiled with Csc, the archive will be included in the compiler command line.

5.3.6 Optimizing and Stripping the Script Binary

Cscc invokes the standard C language compiler "cc". By default, Cscc strips (does not include) the symbol table from the binary and does not optimize the binary. You can change these defaults with the -O, -s, and -g options of the cscc command.

The -O option of cscc specifies that the resulting script object code is optimized. However, since a script is primarily a series of function calls that can not be optimized, the -O option has little effect and slows Cscc operation.

Stripping the symbol table for a script makes it smaller; therefore, executing it requires less memory. However, you can specify that the symbol table is not stripped (is included) in the binary by using the -s option of cscc which invokes the -s option of the "cc" compiler.

If problems occur during script execution, you may want to run a system debugger, such as dbx. System debuggers require that the symbol table is included in the executable script and that the file is not optimized. The -g option of the cscc command invokes the C compiler's -g option to include the symbol table and prevent optimization. This option, described in the manual for the cc compiler, will allow the resulting executable file to be used in a system debugger.

An executable script file created with the -g option will be rather large and will require a large amount of memory to execute. Therefore, using -g is not recommended for scripts executed during a sizable multi-user emulation.

5.3.7 Excluding Help Information from the Script Binary

By default, Cscc includes help information in the executable version of the script it creates. This help information is shown in the Syntax help screen in Section 6.1.

By using the `-h` option of the `csc` command, you may compile the script to exclude help information from the executable file:

```
$ csc -h example30
```

The resulting executable script will be smaller and require a bit less memory to execute.

If you try to display help information for a script that was compiled with the `-h` option, the following error message will result:

```
$ example30 -  
error: example30: no help available. recompile without the -h option.
```

5.3.8 Excluding Monitor Code

By default, Monitor code is included in the executable script file after it has been compiled with `Csc`. An executable script file with Monitor code is somewhat larger requiring more resources to run than a file without the code.

If you want to compile your script without Monitor code and therefore not use the Monitor tool, enter the `-m` option of the `csc` command as demonstrated below:

```
$ csc -m example7
```

5.3.9 The Compiler Command Line

The `-v` option of the `csc` command specifies that the entire compiler command be listed after a script is compiled. By default, EMPOWER/CS lists a condensed version of the compiler command that does not include all libraries and options.

```
cc -s -o script1 ./csccl7366.c /usr/local/lib/*.a
```

```
$ cscsc -v script1
EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

cc -s -L/opt/oracle/lib -cckr -I/usr/local/h -o script1 ./csccl7377.c
/usr/local/lib/empowercsm.a /usr/local/lib/empowercsMON.a /usr/
local/lib/empowerGV.a /usr/local/lib/oralib.a /usr/local/lib/sybstub.a
/usr/local/lib/syb4stub.a -locic /opt/oracle/lib/osntab.o -lsqlnet
-lora -lsqlnet -lcvt6 -lcore -lnlsrtl -lcore -lsocket -lnsl -lm
```

```
csc: can't open example1.c.
```


Deborah Z. Lurie in *Los Angeles*

Two methods are available for executing a script: Non-Display and Display modes. Non-Display mode executes what was captured between the PC and the SUT but involves only the UNIX script driver and the SUT. Display mode involves the PC, the UNIX driver, and the SUT and displays the captured client/server activity on the PC.

If you did not specify `-h` in the `cscd` command, you can access syntax help for a compiled script by entering the script executable name followed by a hyphen (-).

Example:

```
$ example1 -
example1:  EMPOWER/CS V1.0.0, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
Usage:
    example1 [-d hostname] [-S scriptid] [log [arg1 arg2 . . .]]
Options:
    -d hostname      Displays script execution
    -S scriptid      Changes scriptid from example30 if not run from MIX
    log              Identifies the log file to be created
    arg1 arg2 . . .  Identifies optional script arguments

Notes:
    If a log is not specified, the log example1.1 will be created.
    If you specify " " as the log, no log will be created.
    You must specify a log if you want to specify arguments.
    arg1 is accessible as the variable argv[3] in the script.
```

Executing a script in Non-Display mode requires only the UNIX script driver and the SUT. The script on the UNIX driver replaces the PC to interact with the SUT, and the SUT responds as if it is servicing requests from an actual PC.

Executing EMPOWER/CS scripts from the UNIX script driver allows you to set up multi-user emulations by duplicating scripts or combining and executing multiple scripts. Refer to the Multi-User Testing manual for detailed information on executing multi-user emulations with EMPOWER/CS.

To execute a script in Non-Display mode, you must first compile it on the UNIX script driver with the `csc` command.

The script file created by `Csc` is a binary executable file. Therefore, you can execute a compiled script in Non-Display mode by typing the name of the script (with no extension) at the command line with a series of options, listed in the syntax help screen above.

To execute the script `example30`, type the following at the command line:

```
$ example30
```

6.2 Script Execution in Display Mode

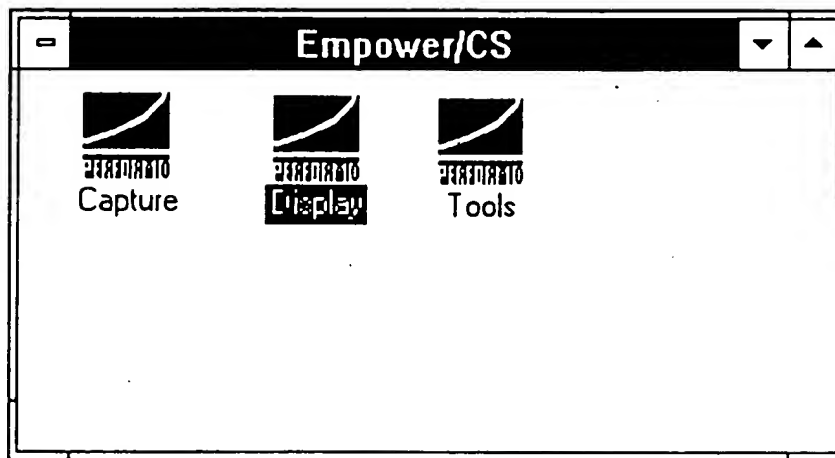
Since the UNIX script driver replaces the PC during script execution, you are not required to display emulated user activities during script execution. However, script development and debugging often are easier if you can see the emulated activity. Therefore, you can display captured user and database activity on one or more PCs as a script executes. Each PC that you wish to display to must have a network connection to the UNIX script driver and to the SUT.

In Display mode, script execution is initiated from the UNIX script driver. The UNIX driver directs the PC to replay captured user activity and to wait for data returned from the SUT. The PC replays the entire script interacting with the SUT as directed by the UNIX script driver. During script execution, all interaction occurs between the UNIX script driver and PC, and between the PC and the SUT. The SUT receives all PC transmissions and responds accordingly to the PC.

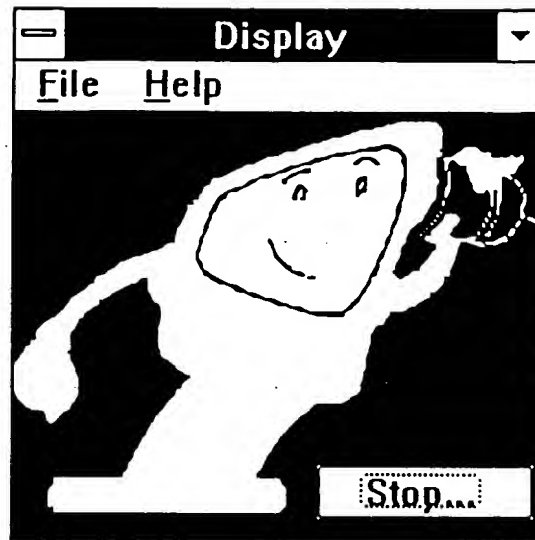
Before a script can be replayed, or executed, in Display mode, you must compile the script on the UNIX driver machine with the `csc` command.

You also must set up the PC to display the script. If your Windows application is not already running, from DOS on the PC, type "win" to start MS Windows.

When the Program Manager appears, select the EMPOWER/CS program group. Then, select the Display program-item icon:



The following window will open that displays a cartoon PC "listening" for network connections from the UNIX script driver:



The UNIX driver must connect to the PC to execute the script and control the captured PC activity. On the UNIX script driver, enter the executable script name with the `-d` option and the name of the PC to which you are displaying.

Example:

```
$ script1 -d vagrant
```

Every user interaction that was captured between the PC and the SUT will execute on the PC. The UNIX driver directs the PC to activate windows, select buttons, enter queries, etc. as if a real user were using the PC. The PC in turn will interact with the SUT as it would if a real user were using it. The SUT will respond to the PC accordingly.

The Display icon on the Windows screen will disappear during script execution and will return only when the script exits.

As soon as the script has completed execution, the computer cartoon will appear again "listening" for any other connections. If you have completed script execution in Display mode, simply select **Stop** in the Display window.

6.3 Script Execution Options

The options listed in the syntax help screen are described in the following sections.

6.3.1 -d

This option is used only for Display mode script execution. On the UNIX script driver, enter the executable script name with the **-d** option and the name of the PC to which you are displaying.

Example:

```
$ script1 -d vagrant
```

See Section 6.2 above for further Display mode instructions.

6.3.2 Changing the Script ID

When you execute a compiled script, a variable called `scriptid` is defined as the name of the script. If you execute a script called `query1`, the variable `scriptid` would be given the value, `query1`.

With the **-s** option of the script execution command, you can change the value of `scriptid`.

Example:

```
$ example1 -S example2
```

The `scriptid` variable is used in Monitor and can be used within a script. If you are running a multi-user test that executes a single script several times simultaneously, you should change the Script ID for each instance of the script. The Script ID acts as an emulated user ID.

Note: If you are using the Mix tool, you do not need to specify the `-s` option to execute a multi-user test. Mix inserts the `-s` option automatically by reading the Script ID from the Mix table. (Refer to the *Multi-User Testing* manual for more information on the Mix table.)

6.3.3 Specifying a Log File

When a script executes, a log file is created in the current directory on the UNIX driver. This log file contains all emulated user activity, SQL requests to the SUT, and all data returned to the PC. To calculate response times, this log file also includes time stamps for the time that such functions as `Beginfunction()`, `Endfunction()`, `Beginscenario()`, and `Endscenario()` were executed.

If you do not specify a name for the log file in the script execution command, the log file will be given the same name as the script with a ".1" extension. You do not need to include the ".1" extension if you specify a log file name.

Example:

```
$ example30 -d vagrant log
```

When you specify a log file name, you can include a path name to save the file in a directory other than the current directory.

Example:

```
$ example30 -d vagrant /user/empower/logs/log001
```

You also can define the E_LOGDIR environment variable to specify a directory for log files as shown below:

In the Bourne shell:

```
$ E_LOGDIR=/user/empower/logs;export E_LOGDIR
```

In the C shell:

```
$ setenv E_LOGDIR "user/empower/logs"
```

If you do not wish to create a log file, specify the null string ("") as part of the script execution command which is useful for performing demos when disk space is limited. If you do not create log files, you will not be able to obtain statistical information for your test.

Example:

```
$ example30 -d vagrant ""
```

6.3.4 Specifying Arguments

Arguments may be passed to the script during execution by specifying them in the script execution command. You must specify log file names prior to specifying arguments, even if you wish to use the default log file name.

Example:

```
$ example30 example30.1 john pass001
```

Refer to Section 7.3 of this manual for more information on script arguments.

6.4 The Log File

Executing EMPOWER/CS scripts results in the creation of a log file for each executed script. If a script is executed so that the created log file has the same name as an existing log file, the new log file will overwrite the existing one. If you are running a multi-user test, you must ensure that each emulated user writes to a different log file.

The log file contains copies of user entries, every EMPOWER/CS function entered, SQL requests to the SUT, data returned to the PC, and time stamps used for performance measurement. Entries in the log file that correspond to lines in the script source file include the characters >>> followed by the line number of the corresponding script file entry.

The following is an example EMPOWER/CS log file:


```

>>>      Log: script1.1
>>>      Date: Fri Jan 20 10:03:02 1995
>>>      Command: script1 -d ergy
>>>      Beginsource("script1")
>>>      4 Typerate(5.00)
>>>      5 Pointerrate(150.00)
>>>      6 Thinkuniform(1.000,2.500)
>>>      7 Seed(28310)
>>>      8 Timeout(300,CONTINUE)
>>>      9 Dberror(CONTINUE)
>>>     10 Unset(NOTIFY)
>>>     12 Beginscenario("script1") 10:03:02.89
>>>     14 InitialWindow(0,"Desktop",0,0,1024,768)
>>>     15 InitialWindow(1,"Program Manager",943,162,186,581)
>>>     16 InitialWindow(2,"Accounting",413,679,1029,771)
>>>     18 LeftButtonPress(448, 692)
>>>     21 AppWait(0.05)
>>>     22 WindowRcv("CwPtPt")
CwPtPt
>>>     24 LeftButtonPress(459, 616)
>>>     26 WindowRcv("DwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt")
SfSfAcPtDwCoCwCwCwCwCwCwCwCwCwCwSf
>>>     28 CurrentWindow("Run",429,491,880,764)
>>>     30 Think 1.923
>>>     33 LeftButtonPress(470, 575)
>>>     35 Type("c:\acct\acct^M")
>>>     37 AppWait(0.22)
>>>     38 WindowRcv("CwCwCwCwCwCwCw")
>>>      Username(LOG1, "scott/tiger@SUT");
>>>      Password(LOG1, "");
>>>      Logon(ORACLE, LOG1);
CwCwCwCwCwCwCw
...

>>>     97 Think 2.141
>>>     98 ButtonPush("Employee Records|Next",739,438)
>>>     99 ButtonPush("Employee Records|Next",739,438)

```

(continued on following page ...)

```
>>> 100 ButtonPush("Employee Records|Next",739,438)
>>> 101 ButtonPush("Employee Records|Close",718,157)
>>> 103 WindowRcv("SfPtDwAcSf")
SfAcSfPtDw
>>> 105 CurrentWindow("Accounting Application",189,82,685,449)
>>> 107 Think 1.069
>>> 109 LeftButtonDown(204, 105)
>>> 110 LeftButtonUp(211, 241)
>>> 113 AppWait(0.39)
>>> 114 WindowRcv("ScDwAcSf")
AcSf
>>>      Commit(LOG1);ScDw
>>> 116 CurrentWindow("Accounting",413,679,1029,771)
>>>      Close(CUR1);
>>>      Logoff(LOG1);
>>> 125 WindowRcv("Pt")
Pt
>>> 127 Endscenario("script1") 10:04:20.11
>>>      Endsource()
```

The UNIX script driver drives the PC, instructing it to activate buttons and keys, to draw windows, to enter data, etc. according to functions in the script. In the above script, those types of user interactions are marked with line numbers which refer back to a particular line in the script .c file. Because this script was executed on the PC in Display mode, you will notice that some lines in the log file contain no numbers corresponding to lines in the source script.

When a script is executed in Display mode, database traffic is generated by the PC, not the UNIX script driver. This traffic is captured and sent to the UNIX driver to be logged in the log file, but because these database functions were not executed by the UNIX driver, they contain no line numbers.

If the script had been executed in Non-Display mode from the UNIX driver, its log file would include line numbers with the database functions because the UNIX driver interacts with the SUT just as a PC would. An example follows:

(continued on following page...)

```
>>> 109 LeftButtonDown(204, 105)
>>> 110 LeftButtonUp(211, 241)
>>> 113 AppWait(0.39)
>>> 114 WindowRcv("ScDwAcSf")
>>> 116 CurrentWindow("Accounting",413,679,1029,771)
>>> 118 Commit(LOG1)
>>> 121 Close(CUR1)
>>> 122 Logoff(LOG1)
>>> 123 Closenv(ORACLE)
>>> 125 WindowRcv("Pt")
>>> 127 Endscenario("script1") 10:12:55.53
>>>      Endsource()
```

6.5 What Comes Next?

Now that you have learned the basic concepts for emulating captured activity by executing a script, you are ready to begin a multi-user emulation. However, before proceeding to the *Multi-User Testing Manual*, you may wish to edit or enhance your scripts to emulate more realistic loads on your client/server system. The following section, Script Content and Enhancement, discusses common script functions, methods for editing your scripts, and advanced emulation techniques that allow you to develop realistic scripts.

```

/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

TypeRate(5);          /* Typing delay in CPS */
PointRate(150);       /* PointRate in PPS */
ThinkUniform(1,2.5);  /* Think delay */
Seed(getpid());       /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if DB transaction takes too long */
DbError(CONTINUE);    /* What to do on Database errors */
Unset(NOTIFY);        /* Don't display warnings. I'll use Mon to find them */

BeginScenario("script1");

InitialWindow(0, "Desktop", 0, 0, 1024, 768);
InitialWindow(1, "Program Manager", 943, 162, 186, 581);
InitialWindow(2, "Accounting", 413, 679, 1029, 771);

LeftButtonPress(448, 692);

AppWait(0.05);
WindowRcv("CwPtPt");

```

(continued on following page ...)

```

LeftButtonPress(459,616);

WindowRcv("DwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");

CurrentWindow("Run",429,491,880,764);

Think(2.53);

/* Clicked (Edit) */
LeftButtonPress(470,575);

Type("c:\\acct\\acct^M");

AppWait(0.22);
WindowRcv("CwCwCwCwCwCwCw");
Openenv(ORACLE1,VERSIONV6V7);

Username(LOG1, "scott/tiger@SUT");
Password(LOG1, "tiger");
Logon(ORACLE1, LOG1);

WindowRcv("SfAcSfDwPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");

CurrentWindow("Accounting",413,679,1029,771);

Open(LOG1,CUR1);

WindowRcv("AcSf");

CurrentWindow("Accounting Application",189,82,685,449);

Think(4.89);
ButtonPush("Accounting Application|Customers",343,288);

AppWait(5.72);
WindowRcv("SfPtCwCwCwCwCwCwCwCwCwCwCwCwCwCwCw");

Begintimer("Accounting Application_Customers");

Dbset(CUR1,DEFER,TRUE);

Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1, ADDRESS_LINE_2,
ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS
FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1,MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);

```

(continued on following page...)

(continued on following page...)

```
Close(CUR1);  
Logoff(LOG1);  
Closenv(ORACLE1);  
  
WindowRcv("Pt");  
  
Endtimer("Accounting");  
  
Endscenario("script1");
```

7.1.1 General Script Functions

Descriptions of general EMPOWER/CS script functions are presented in the following sections.

7.1.1.1 Begin/End Functions

EMPOWER/CS has three sets of "Begin" and "End" functions which mark the start and finish of scenarios and functions. These functions are `Beginscenario()`, `Endscenario()`, `Beginfunction()`, `Endfunction()`, `Begintimer()`, and `Endtimer()`.

The parameter for each function is the name of the script section being defined. Each "Begin" function must have a corresponding "End" function and the parameter used in each "Begin" function must match the name used in the "End" function.

The `Beginscenario()` and `Endscenario()` functions cause scenario time stamps to be recorded in the log file. These functions are inserted into the script .c file automatically during Capture to mark the beginning of the script. Scenarios typically define large sections of emulated activity. Usually an entire script represents a scenario.

The `Beginfunction()` and `Endfunction()` functions are inserted by the EMPOWER/CS user during Capture or during script editing to define specific activity as a C function. These functions cause function time stamps to be recorded in an executed script's log file which allows the Report tool to calculate response time statistics for the function. The Report tool will indicate the response time for executing the entire set of activities in the function.

The `Begintimer()` and `Endtimer()` functions are inserted into the script automatically during Capture if the **Insert Timer** option is specified or manually by the EMPOWER/CS user. They are used to calculate response times for the activities they define. If **Insert Timer** is selected, `Begintimer()` and `Endtimer()` are placed around database traffic that occurs between two user events in the script. If **Insert Timer** is not selected, the user can insert these functions to specify activity to be measured in the script.

7.1.1.2 Typerate and Pointerrate

A high fidelity emulation of PC user activity should include a simulation of the time required to type at the keyboard. EMPOWER/CS allows you to specify a typing speed, measured in characters per second, that is applied to all emulated keyboard activity.

The `Typerate()` function sets the typing speed. During script execution, each time that the emulated user is supposed to be typing at the keyboard, the script will pause a length of time defined as the number of characters to be typed times the type rate specified. For example, if the emulated user must type twelve characters and the type rate is specified as six characters per second (`Typerate(6)`) then the script will pause two seconds when the emulated characters are to be "typed."

The default function, `Typerate(5)`, is placed at the top of every script created by EMPOWER/CS, specifying a type rate of 5 characters per second. You may change this rate during script editing. If you specify a type rate of zero in the script (`Typerate(0)`), then no delay is used when the script emulates a user typing at the keyboard.

EMPOWER/CS applies the type rate by inserting a delay between each emulated typed character listed in the `Type()` function. For example, if the emulated type rate is five characters per second and five characters are to be sent to the SUT, EMPOWER/CS will pause 1 second between each character.

The type delay is applied using UNIX system functions particular to your UNIX machine. Each delay technique may include some inaccuracy of up to 0.5 seconds. This inaccuracy over the course of a type-intensive script may affect the duration of script execution, the actual type rate, and ultimately, the accuracy of the performance statistics.

EMPOWER/CS keeps track of the difference between the specified type rate and the actual type delay. This difference is known as typing drift. Throughout script execution, EMPOWER/CS compensates for typing drift by adding to or reducing type delays as appropriate.

Other supported delay methods on your UNIX machine are `NSDELAY`, `SLEEPDELAY`, `PSEUDODELAY`, and `SELECTDELAY`. `NSDELAY` is accurate to 0.01 seconds. The `SLEEPDELAY` method uses the `sleep(2)` system call, which is accurate only to 0.5 seconds. `PSEUDODELAY` allocates a pseudo port on the machine for delays. It is accurate to 0.1 seconds. If the machine runs out of pseudo ports, the delay is applied with `SLEEPDELAY`. `SELECTDELAY`, which is used for BSD systems, uses the `select(2)` system call and is accurate to 0.01 seconds. Although other delay methods are available, the most accurate delay method on your UNIX script driver is that used by EMPOWER/CS.

An accurate representation of PC user activity also should include a simulation of the time required to move the mouse. EMPOWER/CS allows you to specify a speed for mouse movements, measured in mouse points per second, that is applied to all emulated mouse activity.

The `Pointerrate()` function specifies the points per second that the mouse pointer moves. During script execution, each time that the emulated user is supposed to move the mouse, the script will pause a length of time defined as the number of points moved times the pointer rate specified. For example, if the emulated user points the mouse to 120 locations and the pointer rate is specified as

action occurs. The parameter of the `Think()` function specifies the number of seconds elapsed.

During script execution, `Thinkactual()` tells the script to use values in the `Think()` functions that were captured during your Capture session. The following example demonstrates using this function within a script. When you edit your script, insert `Thinkactual()` as shown below:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

Type(5);           /* Typing delay in CPS */
Pointerrate(150);  /* Pointerrate in PPS */
Thinkactual();     /* Think delay */
Seed(getpid());    /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if function takes too long */
Dberror(CONTINUE); /* What to do on Database errors */
Unset(NOTIFY);     /* Don't display warnings. I'll use Mon to find them */

Beginscenario("example1");
```

Think time distribution may be defined with any of these three think time functions: `Thinkconstant()`, `Thinktne()`, and `Thinkuniform()`. Varying think time distribution throughout a script is common and, typically, each application will have its own think time distribution. These functions define a distribution of randomly chosen think time amounts and can be inserted more than once throughout the script file when editing your script. Because the script was captured with think time values, a value must be specified in `Think()` functions for script execution (even if the value is zero), regardless of the think time distribution.

`Thinkconstant()` specifies that the think time is the same every time a `Think()` function is encountered. The think time amount is specified by the parameter `constant`. The syntax is:

`Thinkconstant(constant)`

`Thinktne()` defines a truncated negative exponential distribution of think time amounts. The parameters specify the minimum, average, and maximum:

`Thinktne(min, avg, max)`

`Thinkuniform()` defines a uniform think time distribution. Think time amounts will be chosen so that they are evenly distributed between the amounts specified by the parameters `min` and `max`:

`Thinkuniform(min, max)`

Random number generation for EMPOWER/CS functions is provided by a random number generator internal to the EMPOWER/CS software. Random number generation is required for the EMPOWER/CS functions `Range()`, `Thinkuniform()`, and `Thinktne()`. The EMPOWER/CS internal random number generator will provide consistent results for all UNIX platforms.

If the same range parameters and seed value are provided for `Range()`, `Thinkuniform()`, and `Thinktne()`, the same random number will be generated every time. To ensure unique random numbers, you should use the `Seed()` function to seed the random number generator uniquely for each user. The example script at the beginning of this section shows the use of the `Seed()` function where the process ID is used as a seed for the random number generator.

7.1.1.4 Timeouts and Database Errors

The `Timeout()` function specifies how long EMPOWER/CS will wait to receive an expected response during script execution. When a matching response is not received in the specified time, the script is said to "time out." The `Timeout()` function specifies the length of time to wait before a timeout and the action to be taken when the script times out. The syntax for the function is:

`Timeout(n, cond)`

Similarly, the `Dberror()` function specifies the action to be taken if a database error occurs. The syntax is:

`Dberror(cond)`

The parameter `n` in the `Timeout()` function is the number of seconds to wait before a timeout (an integer). The `cond` parameter in both `Timeout()` and `Dberror()` is the action to be taken at timeout or database error. This parameter can be either `CONTINUE` or `EXIT` or a user-defined function. `CONTINUE` specifies that script execution continues to the next function in the script and is generally the condition specified. `EXIT` halts script execution.

The default functions, `Timeout(300, CONTINUE)` and `Dberror(CONTINUE)`, are placed at the top of every script created by EMPOWER/CS. During script execution, `Timeout(300, CONTINUE)` specifies that the script will wait 300 seconds (five minutes) for the expected response, then continue. `Dberror(CONTINUE)` specifies that the script will continue if it encounters a database error. You can edit these functions and/or insert them throughout the script to suit your testing needs.

The following conditions may cause timeouts or database errors to occur during script execution:

- If a script times out and continues prematurely, receipt of the "late" response (meaning the response took longer than the specified 300 seconds) likely will throw script synchronization off, which could cause continual timeouts until the script exits. If your SUT is slow, you may need to increase the timeout value.
- When multiple clients are accessing the SUT during script execution, responses to your script(s) or PC (in Display mode) could be delayed.
- During script execution in Display mode, EMPOWER/CS waits for specific `WindowRcv()` patterns. If the expected `WindowRcv()` patterns are not received in the specified time, a timeout will occur.

Valid options are listed below:

<u>Option</u>	<u>Description</u>
LCMD	log miscellaneous commands
FLUSH	flush SUT responses to the log that are detected after a pattern match in <code>s WindowRcv()</code>
NOBUF	don't use buffered writes to the log file
NOTIFY	display a message when a timeout occurs, execution is suspended, or execution resumes
BELL	ring the bell twice when a timeout occurs
LOGGING	Enable logging

7.1.1.6 Mouse Activity

The following list shows all the functions used to emulate mouse button activity for the left, middle, or right mouse buttons during script execution:

<code>LeftButtonDown(x,y)</code>	<code>MiddleButtonDown(x,y)</code>	<code>RightButtonDown(x,y)</code>
<code>LeftButtonUp(x,y)</code>	<code>MiddleButtonUp(x,y)</code>	<code>RightButtonUp(x,y)</code>
<code>LeftButtonPress(x,y)</code>	<code>MiddleButtonPress(x,y)</code>	<code>RightButtonPress(x,y)</code>
<code>LeftDblPress(x,y)</code>	<code>MiddleDblPress(x,y)</code>	<code>RightDblPress(x,y)</code>

These functions are inserted into the script `.c` file during Capture when the specified mouse activity is performed.

The function parameters `x,y` indicate the `xy` coordinates of the mouse on screen at the time the mouse button was activated during Capture.

The "ButtonDown" functions indicate when the specified button on the PC mouse was pressed down.

The "ButtonPress" functions indicate when the specified button on the PC mouse was pressed down and released. A ButtonPress event is inserted in the script when a ButtonDown and ButtonUp event do not contain other user events between them and the x,y coordinates are the same.

The "ButtonUp" functions indicate when the specified button on the PC mouse was released. This function will be captured into the script file instead of a ButtonPress function if the x,y coordinates are different between two consecutive down/up mouse activities.

The "DbIPress" functions indicate that the user double-clicked a mouse button. During script execution, these functions emulate two consecutive presses and releases of the specified button.

During script execution in Display mode, these functions are used to emulate the specified mouse activity. In Non-Display mode, the x,y coordinates are used to simulate moving the mouse by emulating a mouse pointer delay to the next x,y coordinates.

Mouse button events may be edited in your script file, but because you change the activity from when it was captured, you run the risk of breaking the script.

The following example script segment contains various mouse button activities:

```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);
```

(continued on following page...)

```
AppWait(0.38);
WindowRcv("Pt");

LeftButtonDown(340,216);
LeftButtonPress(333,219)

AppWait(0.17);
WindowRcv("Pt");

LeftButtonPress(350,202);
```

7.1.1.7 Keyboard Activity

The following list shows the functions used to emulate keyboard activity during script execution:

KeyPress(key)	SysKeyPress(key)
KeyDown(key)	SysKeyDown(key)
KeyUp(key)	SysKeyUp(key)

The parameter *key* represents a Microsoft Windows virtual key code value. All possible virtual key code values are listed below:

VK_TILDE	VK_LAPPOST	VK_UNDERSCORE
VK_HYPHEN	VK_PLUS	VK_EQUAL
VK_LCURLY	VK_LSQUARE	VK_RCURLY
VK_RSQUARE	VK_COLON	VK_SEMI
VK_DQUOTES	VK_SQUOTES	VK_LTHAN
VK_COMMA	VK_GTHAN	VK_PERIOD
VK_QUESTION	VK_SLASH	VK_PIPE
VK_BKSLASH	VK_CANCEL	VK_CLEAR
VK_TAB	VK_BACK	VK_SHIFT
VK_CONTROL	VK_MENU	VK_PAUSE
VK_CAPITAL	VK_ESCAPE	VK_SPACE

VK_PRIOR	VK_NEXT	VK_END
VK_HOME	VK_LEFT	VK_UP
VK_RIGHT	VK_DOWN	VK_SELECT
VK_EXECUTE	VK_SNAPSHOT	VK_INSERT
VK_DELETE	VK_HELP	VK_NUMPAD0
VK_NUMPAD1	VK_NUMPAD2	VK_NUMPAD3
VK_NUMPAD4	VK_NUMPAD5	VK_NUMPAD6
VK_NUMPAD7	VK_NUMPAD8	VK_NUMPAD9
VK_MULTIPLY	VK_ADD	VK_SEPARATOR
VK_SUBTRACT	VK_DECIMAL	VK_DIVIDE
VK_F1	VK_F2	VK_F3
VK_F4	VK_F5	VK_F6
VK_F7	VK_F8	VK_F9
VK_F10	VK_F11	VK_F12
VK_F13	VK_F14	VK_F15
VK_F16	VK_F17	VK_F18
VK_F19	VK_F20	VK_F21
VK_F22	VK_F23	VK_F24
VK_NUMLOCK	VK_SCROLL	

These functions are inserted into the script during Capture when the specified key activity is performed.

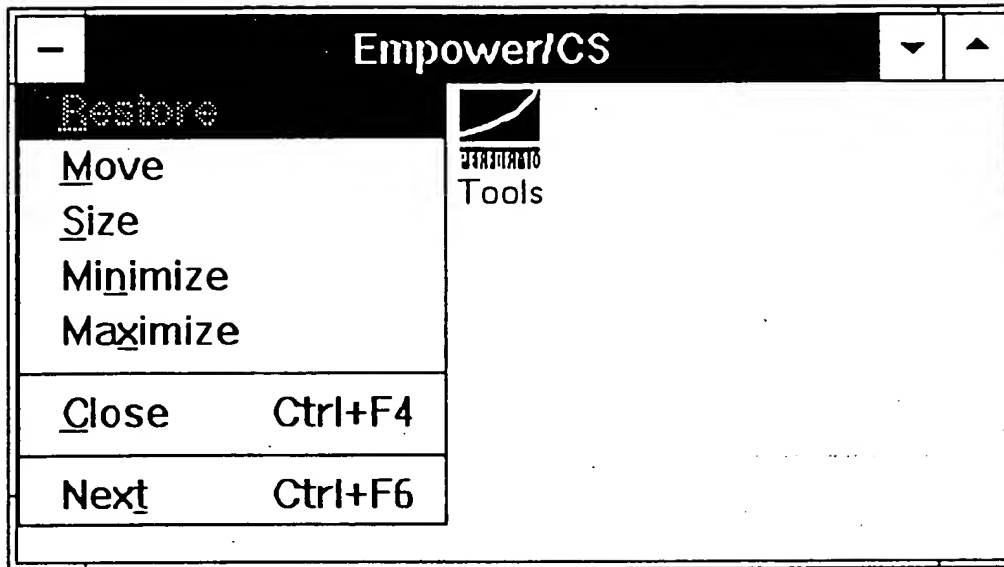
The "KeyDown" functions indicate that a non-printable keyboard key on the PC was pressed down.

A "KeyPress" function is translated into a KeyDown/Up sequence. A KeyPress function is captured into the script when no user events are contained within a KeyDown/Up pair.

The "KeyUp" functions indicate that a non-printable keyboard key on the PC was released.

The "SysKey" functions indicate that the keyboard activity included pressing the Alt key. "Sys" implies that the keystrokes are being sent to the System Menu of

the current window which is located under the icon in the top left corner of the window as shown below:



During script execution in Display mode, these functions are used to emulate the specified keyboard activity. In Non-Display mode, type rate is applied to these events and the script delays accordingly.

Key events may be edited in your script file, but because you change the expected activity from when it was captured, you may break the script when it is executed.

The following example demonstrates various keyboard and mouse activity in a script file:

```

Beginscenario("example2");

LeftButtonPress(188,381);

WindowRcv("CwPtPtDwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");

```

(continued on following page...)

```
LeftButtonDown(193,316);
LeftButtonUp(193,317);

CurrentWindow("Run", 189,82,685,449);

AppWait(0.11);
Type("c:\\acct\\c\\acct^M");

WindowRcv("CwAcSfDwPt");

CurrentWindow("General Ledger", 234,99,704,520);

AppWait(1.26);

SysKeyDown(VK_MENU);
SysKeyPress(VK_SPACE);
SysKeyUp(VK_MENU);

Type("scott^Itiger^IFOO^M")
```

7.1.1.8 Type

The `Type()` function is recorded into a script file during Capture when keystrokes are entered from the keyboard. The parameter of the `Type()` function may include the standard keyboard keys pressed (except for those defined as virtual key codes in Section 7.1.1.2) and the non-displayed key combinations defined below. Each key translates into a KeyDown/Up pair for the emulation.

Non-displayed key combinations are shown in the parameter of the `Type()` function using the standard UNIX technique in which the character `^` represents the Control key. Common control sequences captured into the `Type()` function are listed below:

- `^M` Carriage Return
- `^H` Backspace
- `^I` Tab
- `^?` Delete

Note: You may notice that if you use the number pad or a symbol font during your Capture session, the numbers or symbols will not be captured into the `Type()` function. For instance, if you are using a symbol font where the "A" key represents "α", pressing this key will be captured as `Type("a")`. To capture numerical characters, you should use those listed at the top of the standard keyboard. Only standard keyboard keys are captured into a script .c file.

During script execution in Display mode, the `Type()` function is used to emulate a user typing the specified keystrokes. In Non-Display mode, this function emulates a typing delay.

Type strings are very easy to modify and can be edited to alter the keystrokes typed by an emulated user. For example, the following `Type()` function is entered in a query:

```
Type("13402^M");
```

It can be modified to:

```
Type("98704^M");
```

The following example shows a portion of a script containing a `Type()` function in which the user entered a command during Capture to run an application:

```
AppWait(0.11);  
  
Type("c:\\acct\\c\\acct^M");  
  
WindowRcv("CwAcSfDwPt");  
  
CurrentWindow("General Ledger",189,82,685,449);
```

Note: If executing your script in Non-Display mode, any modifications you make to the `Type()` function will not affect the data that is input to the database. It will only affect the amount of type delay to be emulated. If you do want different data to

be input to the database, you will need to change the corresponding `Parse()` or `Data()` statements.

7.1.1.9 ButtonPush

The `ButtonPush()` function is inserted into the script file during Capture to indicate that a specific button, a MS Windows pushbutton such as **OK**, **Cancel**, **Yes**, **No**, etc., was activated. It is used during script execution in Display mode to move the mouse to activate the specified pushbuttons. In Non-Display mode, this function simulates mouse movement as defined in the `x,y` parameters to allow for mouse pointer delay.

The syntax of this function is:

```
ButtonPush(str, x,y);
```

The parameter `str` may be listed in a format similar to the following examples:

```
ButtonPush("Program Manager|Run|OK",226,99);
```

```
ButtonPush("Tools|#c1|#c4",68,69);
```

```
ButtonPush("OK", 187, 201);
```

```
ButtonPush("#c2",299,134);
```

The format of the `str` parameter is designed so that EMPOWER/CS can easily locate the captured pushbutton during script execution in Display mode. This format is based on the MS Windows concept of a tree structure.

The MS Windows tree structure is based on a heirarchy of windows where each window that is accessed from a primary, or parent, window is a child of that parent. The `str` parameter is pipe-delimited, and is listed right to left from child to parent window where the right-most item is the name of the pushbutton. If one of the windows or the pushbutton has no title, something like "#c1" will be listed to

began. Therefore, if multiple applications are open when you begin Capture, multiple `InitialWindow()` functions will be listed in the script file.

The following example demonstrates `InitialWindow()` functions captured in a script file:

```
Beginscenario("script1");  
  
InitialWindow(0,"Desktop",0,0,1024,768);  
InitialWindow(1,"Program Manager",989,34,232,453);  
InitialWindow(2,"MS-DOS Prompt",0,0,MAXWIDTH,MAXHEIGHT);
```

The first parameter of the `InitialWindow()` function indicates the position of the program in the MS Windows task list. The program's position in the Windows task list is important for users who capture a script using "Fast-Tab" task switching. Position 0 is a reserved value; it is used to record the screen resolution during Capture and verify the resolution for script execution in Display mode.

The second parameter identifies the title of the window.

The next two numerical parameters indicate the x,y coordinates of the top left portion of the window on screen. Then, the next two parameters indicate the width and height of the window. If the width and height are both zero, the window is minimized. If the x,y coordinates are both zero, and width is `MAXWIDTH` and height is `MAXHEIGHT`, the window is maximized. If neither of these conditions apply, the window is in a normal state.

During script execution in Display mode, `InitialWindow()` attempts to locate the windows listed as parameters and place them in their captured positions. The function does not apply to Non-Display mode script execution.

If the specified programs are not open at the time of script execution or if a window could not be moved to the specified position, the log file will record a warning but script execution will continue. Therefore, before you execute your script in Display mode, your desktop should have the same applications open as when you captured the script.

You should not attempt to remove or edit this function in your script file because you would change the state of the Windows desktop from when it was captured, and, therefore, may break the script.

Note: The first `InitialWindow()` function listed in the script indicates the resolution of the screen as the first xy parameters are 0,0. You will notice a task order of 0 specified in this first `InitialWindow()` function. The zero indicates to EMPOWER/CS to check the screen resolution making sure the script is displayed in the same resolution as captured. For instance, if a script was captured in VGA mode and you attempted to display it in SuperVGA mode, the captured xy coordinates would be invalid. Therefore, when displaying your script, you should use a screen that is the same resolution as the screen used during Capture.

7.1.1.11 AppWait Delay

The `AppWait()` function is inserted into your script file during Capture before a `WindowRcv()` function to record the amount of time taken to draw a window on screen. This function is used during script execution to emulate the application delay for drawing a window on screen. During script execution, this function only applies to Non-Display mode, because in Display mode, windows would actually be activated.

In the following example, the script recorded an application delay of 1.16 seconds to draw the "Program Manager" window. During script execution in Non-Display mode, the script will emulate the application delay by pausing the script for the time specified.

The parameter of `AppWait()` is the time in seconds of the captured application delay. If you wish to change your AppWait delay, you may set a multiplication factor with the `AppWaitFactor()` function. This function multiplies the AppWait delay times the factor. For instance, during script execution your application may be twice as slow because it is receiving twice as much data from the SUT. You can set the `AppWaitFactor()` to .2 so that the script will emulate the extra delay.

Example:

```
AppWaitFactor(0.20);
AppWait(1.16);
WindowRcv("SfDwAcSfSfSfPt");
```

7.1.1.12 WindowRcv

The `WindowRcv()` function is inserted into the script during Capture when an activated window is drawn on screen and usually follows an `AppWait()` call. Consecutive `WindowRcv()` functions can be listed within the script file.

This function is used during script execution in Display mode to ensure that the activated window is drawn on screen. In Non-Display mode, this function is ignored because the `AppWait()` function emulates the amount of time required to draw the specified window.

The parameters of this function are MS Windows standard two-letter pneumonics that represent the following commands:

<u>Command</u>	<u>Description</u>
Ac	Activate Window
Co	Command
Cw	Create Window
Dw	Destroy Window
Pt	Paint
Sf	Set focus
Sz	System Command

Example:

```
AppWait(0.05);  
WindowRcv("CwPtPtDwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");  
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");  
  
CurrentWindow("Run", 429, 491, 880, 764);
```

This function should not be edited in your script file when displaying script execution because you change expected activity from when it was captured and, therefore, run the risk of breaking the script.

7.1.1.13 CurrentWindow

The `CurrentWindow()` function is inserted into your script file to record all titled windows that were activated during the Capture session. This function is used to ensure windows are in their captured state during script execution in Display mode. It will be listed in Monitor during script execution in Display and Non-Display modes adding context to script execution.

The parameters of `CurrentWindow()` identify the title of the current window (window with focus), define the top left xy coordinates, and then, define the width and height of the window's position on screen. If the width and height are both zero, the window is minimized. If the x,y coordinates are both zero and width is `MAXWIDTH` and height is `MAXHEIGHT`, the window is maximized. If neither of these conditions apply, the window is in a normal state.

During script execution, if the window could not be moved to the specified position, the log file will record a warning but script execution will continue.

The following example demonstrates how `CurrentWindow()` would appear in a script file. The activated window was "Accounting" which appeared on screen in a normal state:

```
AppWait(0.39);  
WindowRcv("ScDwAcSf");  
  
CurrentWindow("Accounting",413,679,1029,771);
```

The following example is a portion of a log file that recorded a warning when a `CurrentWindow()` function could not execute properly because the window did not exist during script execution in Display mode:

```
>>> CurrentWindow("File Manager",0,0,MAXWIDTH,MAXHEIGHT)  
Warning: Unable to find window
```

This function should not be edited or removed from your script file because you change the state of your Windows desktop from when it was captured and, therefore, run the risk of breaking the script.

7.1.2 Interacting with the SUT

For you to fully understand the process of interacting with the SUT and the EMPOWER/CS functions associated with this process, the general behavior of the database environment should first be explained. The following paragraphs and sections will introduce you to levels of the database environment that represent specific database communication structures, define the actual script statements that represent such operations as querying or inserting data, and describe other EMPOWER/CS functions that perform various database operations.

7.1.2.1 Communication Structures

In the database environment, certain communication structures are defined to access the database on the server for retrieving, inserting, or changing data. These

- The first level of the database environment is the database environment as a whole. An entire environment is defined that identifies the database being accessed.
- The next level is a log on communication structure. To connect to the database, at least one log on communication structure is defined. If the client application requires multiple, simultaneous connections, one log on structure is defined for each connection.
- The third level of the database environment is the cursor structure. To process SQL requests to the SUT, a cursor communication structure is defined for each SQL request.

The following diagram demonstrates the database environment structure and how the SUT is accessed.

Diagram illustrating a multi-processor system architecture. The system consists of two processors (top and bottom) and a central component labeled SUT (System Under Test).

Top Processor:

- Local Log: LOG1
- Buffers: CUR1, CUR2

Bottom Processor:

- Local Log: LOG2
- Buffers: CUR3, CUR4

SQL (System Query Language) Arrows:

- Four arrows labeled SQL point from the processors towards the SUT component.

SUT (System Under Test):

- Represented by a vertical bar on the right side of the diagram.

Copyright PERFORMIX, Inc. © 1995

Basic Procedure and Script Structure for Database Activity

- ☐ A database environment is opened, represented by the call `Openenv()` in the script which specifies the database to be accessed
- ☐ A connection to one or more databases occurs with a `Logon()` to each specified database
- ☐ One or more cursors are opened, represented by the `Open()` function in the script, to process SQL statements
- ☐ The SQL statements required to perform database operations such as querying, inserting, or deleting data are processed with `Parse()`
- ☐ The select-list items of the SQL statement are described as necessary with `Describe()` functions
- ☐ `Bind()` functions are called for input statements to bind the address of an input variable to each placeholder in the SQL statement
- ☐ For queries, `Define()` is called to define an output variable for each select-list item in the SQL statement
- ☐ `Exec()` is called to execute the `Parse()` and, therefore, the SQL statements
- ☐ For queries, `Fetch()` and `GetNextRow()` are called to retrieve and sort through specified rows of data
- ☐ The cursors are closed with `Close()`
- ☐ A `Logoff()` function is called for each log on structure to disconnect from the specified database(s)
- ☐ Each database environment is closed with `Closenv()`


```
Type("c:\\accts\\accts^M");

AppWait(0.22);
WindowRcv("CwCwCwCwCwCwCw");
Openenv(ORACLE,VERSIONV6V7);

Username(LOG1, "scott/tiger@FOO");
Password(LOG1, "tiger");
Logon(ORACLE, LOG1);

WindowRcv("SfAcSfDw");

Open(LOG1,CUR1);

WindowRcv("AcSf");

CurrentWindow("Accounts Application",189,82,685,449);

Think(4.77);
ButtonPush("Accounts Application|#c5",355,270);

AppWait(5.38);
WindowRcv("SfCwCwCwCwCwCwCwCwCwCwCwCwCwCwCwCw");
Dbset(CUR1,DEFER,TRUE);

Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1, ADDRESS_LINE_2,
ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS
FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1,MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
```

EMPOWER/CS-V1.0.1

The following sections further explain the procedure for interacting with the SUT.

As stated in the previous section, the first level of the database environment is the database environment as a whole. This level is represented in a script file by such database names as ORACLE, SYBASE2, etc. and is accessed with the function `Openenv()`. The `Openenv()` function is inserted into the script when a database environment is opened. When executing a script, `Openenv()` opens an environment to specify the database and database version to be used for the emulation.

EMPOWER/CS-V1.0.1

Note: In some cases, you also may notice the functions, `AppName()`, `Hostname()`, and `Servername()` before the `Logon()` call in your script. These functions also are used to access the database. `AppName()` specifies the name of the database application being used. `HostName()` specifies the name of the host machine where the server resides. `Servername()` is the name of the server on the host machine where the database is located.

7.1.2.5 Open the Cursors

The third level of the database environment is the cursor communication structure. The cursor structure is where the script actually interacts with the database by processing SQL requests to the SUT. Cursors are opened with an `Open()` call in the script file and are represented in function parameters as a pointer to a cursor structure called `CUR1`, `CUR2`, etc. The first cursor opened in a particular log on structure will be numbered by EMPOWER/CS as `CUR1`, the second as `CUR2`, etc. The `Open()` function is inserted into the script when a cursor structure is opened.

A cursor can be opened only through a successful log on connection and multiple cursors can be opened from a single log on. Processing a SQL statement is possible only from an open cursor structure. Therefore, such script functions as `Parse()`, `Define()`, `Bind()`, `Exec()`, `Fetch()`, `GetNextRow()`, etc. which process the SQL statement operate only from a cursor structure. The SQL statement is a parameter to the `Parse()` functions and specifies the operations of querying, inserting, or deleting data.

The `Open()` function will be listed after the `Logon()` function and before the `Parse()` in your script file.

7.1.2.6 Parse the SQL Statement

Parsing the SQL statement associates it with the appropriate cursor in the script. Parsing a SQL statement includes sending it to the SUT, verifying its syntax is

compatible with the database, and preparing the SUT for subsequent operations specified in the SQL statement. The `Parse()` function is inserted into your script when a SQL statement is parsed and sent to the SUT. Every SQL statement in a script must be parsed with the `Parse()` function. Once a `Parse()` is performed, the parsed statement is stored on the database to wait for an `Exec()` call.

The first parameter of the `Parse()` function identifies the cursor structure that was opened with the associated `Open()` function. The second parameter lists the SQL statement to be parsed.

The SQL statement specifies the set of data that is to be operated on by the database. This statement is a request to the SUT to select, input, delete, update, or retrieve data. Subsequent functions (i.e, `Define()`, `Bind()`, `Exec()`, `Fetch()`) designate how to perform the database operations and reference the SQL statement by listing the associated cursor number in a function parameter.

Because the SQL statement can be executed only from a cursor structure, `Parse()` occurs after an `Open()` function; and, because the SQL statement is executed by the `Exec()` function, `Parse()` occurs before `Exec()` in the script.

In the following example, the SQL statement for the cursor, `CUR1`, begins with "SELECT ". This example demonstrates a `Parse()` function selecting data for a database query:

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER,
COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");
```

The following example demonstrates a sample SQL statement for inserting data:

```
Parse(CUR1, "INSERT INTO CUSTOMERS ( ID, FIRST_NAME, LAST_NAME,
ADDRESS_LINE_1, ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER,
FAX_NUMBER, COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS ) VALUES (
789, 'Robert', 'Adams', '1 Maple Lane', 'Anytown', 'VA, 22222',
'555-9856', '555-7634', 500, 700, 'Good Job' )");
```

The following example demonstrates a sample SQL statement for deleting database records:

```
Parse(CUR1, "DELETE FROM CUSTOMERS WHERE ID = 789 AND FIRST_NAME =  
'Robert' AND LAST_NAME = 'Adams' AND ADDRESS_LINE_1 = '1 Maple  
Lane' AND ADDRESS_LINE_2 = 'Anytown' AND ADDRESS_LINE_3 = 'VA,  
22222' AND PHONE_NUMBER = '555-9856' AND FAX_NUMBER = '555-7634'  
AND COMM_PAID_YTD = 500 AND ACCOUNT_BALANCE = 700 AND COMMENTS =  
'Good Job' ");
```

Note: If you are using an older version of an Oracle database, you may notice the `Sql()` function in your script file instead of `Parse()`. The `Sql()` function is identical to `Parse()` except that it does not allow for a deferred parse. See Section 7.1.3.1 for more information on deferred parses.

7.1.2.7 Describe Select-List Items

If a description of a variable in a SQL query statement was requested by the client application during Capture, a `Describe()` function will be inserted into your script file. The variable is specified in the function's parameter by a cursor number and by the variable's position in the SQL statement. The `Describe()` function will occur in the script after a `Parse()` statement and before `Define()`.

During script execution, `Describe()` sends a request to the database for a description of specified select-list items in the SQL statement. It returns information about the select-list items needed to determine how to convert, display, or store the data that will be returned when rows are fetched for a query. Such returned information can include the name of the variable, the data type, the size of the item, whether the data is null-terminated or updateable, etc.

Note: If you notice the function `DescribeAll()` in your script, a `Describe()` operation was performed describing every select-list item listed between two positions.

Example:

```
DescribeAll(CUR1, 1, 12);
```

In this example, all variables listed in the SQL statement from CUR1, starting from position 1 to position 12, will be described.

In some cases, you may notice a `DescribeProc()` function instead of `Describe()`. The `DescribeProc()` function describes all variables used in a specified stored procedure. A stored procedure is an operation that is stored on the database to be executed later. The procedure must be described before a parse is completed. Therefore, this function will occur in the script before `Open()` and `Parse()`.

7.1.2.8 Bind the Addresses of Input Variables

Some SQL statements require that data be input to the database. Placeholders for input variables in the SQL statement mark where specific data must be input and are indicated by leading colons (Example: `:NAME`). If a SQL statement requires data to be input to the database, a `Bind()` function will be inserted into the script to bind each placeholder to a variable that is to be passed to the database. Therefore, when the SQL statement is executed, the database will receive the data that was placed in the input variables.

The parameters of the `Bind()` function identify the variable by the cursor number of the associated SQL statement, the variable placeholder listed in the SQL statement, the variable's data type, and the variable's length in bytes.

The `Bind()` function is listed in the script after a `Parse()` statement and before `Exec()`.

The following example demonstrates a `Bind()` function inserted into a script for the variable, `x`:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,  
DEPTNO FROM EMP WHERE EMPNO=:X");  
  
...  
  
Bind(CUR1, "X", STRING, 10);
```

Note: If you notice a `Bindp()` function in your script, a `Bind()` operation was performed for a variable based on its position in the SQL statement instead of placeholder name. The parameter would specify a position instead of a name. `Bindp()` binds the variable's position to the variable.

If a variable was inserted and then retrieved, the `BindDefine()` function is inserted into the script. `BindDefine()` inputs a variable to the database and then gets a new value. Usually, the value returned is the old value. For example, if the SQL statement specified that the value `Smith` be inserted into a record to overwrite the existing value `Jones`, `BindDefine()` would specify `Smith` as the new value for the record and then return the old value `Jones` to the variable.

7.1.2.9 Define

The `Define()` function is inserted into the script when output variables are defined for storing data to be fetched from the database. The SUT places data in these output variables when a `Fetch()` function is called. `Define()` functions are used only when fetching records from the database for a query.

`Define()` associates the address of an output variable in the program with each select-list item in a SQL query statement. The `Define()` function defines each select-list item in the SQL statement by a cursor number, the item's position in the SQL statement, the variable's data type, and the variable's length. The positions in the SQL statement are defined beginning with 1 for the first (or left-most) select-list

item, 2 for the second, and so on. The `Define()` function is listed in the script after a `Parse()` statement and before `Fetch()`.

The following script segment demonstrates the `Define()` statements defining variables for each select-list item of the SQL statement. Notice in the first `Define()` statement below, the variable position "1" refers to `ID` in the SQL statement:

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER,
COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1, MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);
```

7.1.2.10 Execute the Parse

The `Exec()` function executes the operations specified in the associated SQL statement. This function is inserted into your script when a SQL statement is executed on the SUT.

`Exec()` executes the SQL statement by specifying the data the script will be fetching or by inputting the values in all bind variables to the database.

The parameter of `Exec()` identifies the cursor of the associated SQL statement.

7.1.2.11 Fetch the Rows of Data for a Query

After the specified output variables are defined and the SQL statement has been executed, the rows of data that satisfy a query are fetched. In your script, the `Fetch()` function is inserted when the data specified in the SQL statement to satisfy a query is retrieved. During script execution, when the number of rows of data specified in the option `FETCHSIZE` is fetched from the database, the data is placed into the defined output variables in a buffer on your UNIX script driver machine. The parameter of `Fetch()` specifies the cursor structure for the associated SQL statement.

The number of rows of data that are fetched is specified in the `Dbset()` function in the following format:

`Dbset(curnum, FETCHSIZE, n)`

The parameter `FETCHSIZE` sets the number, `n`, of rows of data to fetch. See Section 7.1.3.1 for more information on setting the `FETCHSIZE`.

Each `Fetch()` statement returns the next row from the set of rows that satisfies a query. After the last row has been returned, the next fetch will return an error that no remaining rows could be fetched.

The `Fetch()` statement is called only after the `Parse()` and `Exec()` functions have been called.

In the following script segment, `Fetch()` is called for the cursor, `CUR1`:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,  
DEPTNO FROM EMP, FOR UPDATE OF EMPNO, ENAME, JOB, MGR, HIREDATE,  
SAL, COMM, DEPTNO");  
  
...  
  
Exec(CUR1);  
  
Fetch(CUR1);
```

Note: If you notice a `FetchRaw()` function in your script, this function signifies that binary data was specified in the SQL statement to be retrieved. `FetchRaw()` is used because the data is too large to transfer across the network at one time. Instead, the data is transferred in portions at a time. The parameters to `FetchRaw()` identify the cursor number of the associated SQL statement, the position of the associated select-list item in the SQL statement, the data type of the output variable, and the length in bytes of the data to be retrieved.

In your script file you also will notice the function `GetNextRow()` which is inserted after `Fetch()`. The `GetNextRow()` function is inserted into the script when the client application sorts through the fetched rows of data.

During script execution, `GetNextRow()` is used to view the contents of the fetched rows of data one at a time after they are fetched onto the UNIX script driver. This function generally occurs in a loop and may be used to verify that the requested data was retrieved or to locate a specific row of data.

The following example demonstrates calling `GetNextRow()` in a script file:

```
Fetch(CUR1);  
while (GetNextRow(CUR1) != NOMOREROWS);
```

7.1.2.12 Close the Cursor Structures

Before a client application that processed SQL statements exits, each open cursor is closed. Once a cursor is closed, no additional operations can be performed on that cursor.

The EMPOWER/CS function, `close()`, is inserted into the script for each cursor structure that was closed. When an executing script exits, each cursor opened in a particular log on structure is closed with a `close()` function (one for each `open()` call).

7.1.2.13 Disconnect from the SUT

When a client application that processed cursors exits, each connection to the database is closed. The `Logoff()` function is inserted into the script for each connection to the database that was closed. All active cursors on the specified log on structure must be closed before the log on structure is closed.

7.1.2.14 Close the Environment

The function `Closenv()` is inserted into a script file when a database environment is closed.

A `Closenv()` function closes each environment that was opened with each `Openenv()`. After a database environment is closed, no further log on connections can be made within the specified database environment. Each database environment in a script must be closed before an executing script exits.

The following example demonstrates that the database environment ORACLE was closed:

```
Close(CUR1);  
Logoff(LOG1);  
Closenv(ORACLE);  
  
Endscenario("script1");
```

7.1.3 Other Database Functions

The following sections describe other EMPOWER/CS database functions that may appear in your script file.

7.1.3.1 Dbset

Client applications may specify certain options that control the behavior of the database environment. The `Dbset()` function is inserted into the script when such options are set according to the client application and the SUT and how they interact. This function may occur throughout a script and it specifies various options for the database environment.

The syntax for `Dbset()` follows:

```
Dbset(num, opt, value)
```

The parameter `num` will be an environment number, a log on structure, or a cursor number; the parameter `opt` specifies the database option being set; and, `value` represents either a numerical or string value, or `TRUE` or `FALSE`, depending on the option being set.

The `Dbset()` options that commonly will show up in your scripts are defined below:

<u>Option</u>	<u>Description</u>
<code>FETCHSIZE</code>	Specifies the number of records to fetch from the database when a <code>Fetch()</code> function is executed. The <code>Dbset()</code> function that sets the <code>FETCHSIZE</code> will occur before a <code>Fetch()</code> . The <code>FETCHSIZE</code> value can not be larger than the value specified for <code>MAXARRSIZE</code> . If the <code>FETCHSIZE</code> was specified as 50 and <code>MAXARRSIZE</code> was specified as 20, EMPOWER/CS will reduce the <code>FETCHSIZE</code> to 20. The default value for this option is 1 which means that if <code>FETCHSIZE</code> is specified as zero or lower, EMPOWER/CS automatically sets the value to 1.
<code>MAXARRSIZE</code>	Sets the maximum array size for retrieving or inserting records. If the array size is set at 20, 20 rows of data can be inserted or fetched. If this option is set at 20, then 20 placeholders are allocated for inserting or fetching 20 rows of data. The <code>Dbset()</code> function that sets the <code>MAXARRSIZE</code> will occur before the <code>Bind()</code> and <code>Define()</code> functions in a script. The default value for this option is 1 which means that if <code>MAXARRSIZE</code> is specified as zero or lower, EMPOWER/CS automatically sets the value to 1.
<code>INSERTSIZE</code>	Specifies the number of rows of data to be inserted into the database. This option will be listed before the <code>Data()</code> function in a script in the format <code>Dbset(CUR1, INSERTSIZE, n)</code> . This option allows array binding. For example, if <code>n</code> is 0 or 1, one row can be inserted at a time into the database. If <code>n</code> is 50, 50 rows will be inserted into the database and 50 <code>Data()</code> lines must be listed for every row before <code>Exec()</code> . The <code>INSERTSIZE</code> value can not be larger than the specified <code>MAXARRSIZE</code> . The default value for this option is 1 which means that if <code>INSERTSIZE</code> is specified as zero or lower, EMPOWER/CS automatically sets the value to 1.
<code>OFFSET</code>	Specifies the offset for inserting records in an array. This option is used with the <code>INSERTSIZE</code> option. If an array includes four names and the <code>OFFSET</code> is set to 2, the second name will be the starting point for inserting data. The <code>Dbset()</code> function specifying this option will occur before an <code>Exec()</code> function. If the <code>OFFSET</code> is specified as a range larger than the <code>MAXARRSIZE</code> , a database error will occur.

WAITRES Specifies whether or not to wait for resources. If this option is set to **TRUE**, the script will wait indefinitely for requested information from the database. If it is set to **FALSE** and the script does not receive the requested information, the script will receive an error that the resource was not available. Script execution will then either continue or exit based on the condition set in `Dberror()`. The default value of this option is **TRUE**.

DEFER Specifies whether or not to defer the `Parse()` statement. If this option is set to **TRUE**, a deferred parse will be performed when the script encounters the `Parse()` function. If the option is set to **FALSE**, a normal `Parse()` is executed. The default value of this option is **FALSE**.

Normally, the SQL statement is sent over to the database when the script encounters `Parse()` and processed to ensure it is semantically correct. The SQL statement is stored to wait for the `Exec()` call that actually executes it. If **DEFER** is set to true, the SQL statement is not sent over to the database until the script encounters an operation that requires input from the database such as `Exec()` or `Describe()`.

A complete list of `Dbset()` options that may appear in the script file is listed below with brief descriptions. This list is divided into those options that set integer values and those options that specify **TRUE** or **FALSE**. The list also designates the options that apply specifically to the database environment as a whole, to the log on structure, or to the cursor.

Integer Value

<u>Option</u>	<u>Description</u>
FETCHSIZE	number of rows to be fetched
MAXARRSIZE	maximum number of rows to be fetched or inserted
INSERTSIZE	number of rows to be inserted
OFFSET	row number where to begin inserting
MAXCONNECT	maximum number of connections in an environment
PACKETSIZE	maximum packet size on logon connection
ROWCOUNT	maximum number of rows to return
TEXTSIZE	limit on size of text or image data
ISOLATIONLEVEL	transaction isolation level
AUTHOFF	turns specified authorization off

TRUE or FALSE

Database environment only

Database environment or logon structure

Logon structure

EMPOWER/CS-V1.0.1

Cursor.

7.1.3.2 Data

The `Data()` function is used with database operations for inserting or updating data. It specifies the data to be input to the database by listing the data for each input variable specified with `Bind()`. Therefore, it will follow all `Bind()` functions and will occur before `Exec()`. This function is inserted into the script when data is specified to be input to the database.

The `str` parameter is pipe-delimited and lists the data for each input variable that was defined with `Bind()`. The data listed in this `str` parameter will be listed in the same order as it was specified in the `Bind()` functions.

The `Dbset()` option `INSERTSIZE` applies to this function in that the number of `Data()` functions inserted will correspond to the value of `INSERTSIZE`.

Example:

```
Parse(CUR1, SELECT ENAME, EMPJOB, WHERE ENAME=:name...);
...
Bind(CUR1, "NAME", STRING, 5);
...
Data(CUR1, "SMITH");
```

In this example, SMITH is the data to be entered into the database for the cursor, CUR1.

In another example, a SQL statement may include the following:

```
...EMPNO, ENAME, EMPJOB WHERE EMPNO=:empno, ENAME=:name,
EMPJOB=:empjob...
```

A Data() line that refers to this SQL statement may look like the following:

```
Data(CUR1, "123|Smith|typist")
```

123 would correspond to empno, Smith would correspond to ename, and typist would correspond to empjob.

Because all users would not make the same entries to the database, you may edit this function to vary the data that is inserted into the database during multi-user emulations for a more realistic load. You can edit the `Data()` functions in a set of scripts so that each emulated user inserts a different record. For example, if a `Parse()` statement contains `WHERE NAME = SMITH`, and the `Data()` line is `Data(CUR1, "SMITH")`, you could change this `Data()` line to `Data(CUR1, "JONES")` or `Data(CUR1, "ADAMS")` for each emulated user.

7.1.3.3 SqlExec

If you notice a `SqlExec()` function in your script, a SQL statement was executed that contained no input or output variables. The `SqlExec()` function includes four database operations. It opens a cursor, parses a SQL statement, executes the `Parse()`, and closes the cursor. When this function is used, no operations such as `Describe()`, `Bind()`, `Define()`, `Fetch()`, etc. are necessary. For example, it may be used when a database operation only requires joining two tables.

The syntax for this function is:

```
SqlExec(lognum, sqlstmt)
```

The `lognum` parameter designates the log on communication structure and `sqlstmt` lists the SQL statement.

7.1.3.4 Commit, Rollback

The `Commit()` function is inserted into the script when database processing is committed to the database. This function commits to the database all processing the script has completed (i.e., updating records, deleting records, adding records, etc.) since processing was last committed to the database. Its parameter may specify a log on or cursor structure for which operations are committed.

The `Rollback()` function may be inserted into the script if database operations are not committed but are rolled back since data was last committed. `Rollback()` restores the database to its original state since the last commit was executed.

These functions may appear throughout the script dependent on the behavior of the client application and the SUT.

7.1.4 Editing Database Functions

The EMPOWER/CS database functions translate client application and database interaction and are inserted into the script as the associated actions occur. This process ensures that during script execution a load is generated on the SUT as close to the real interactions as possible. Because the EMPOWER/CS database functions are not user-defined but are inserted into your script based on how the client application interacts with the SUT, you generally should not attempt to edit these functions or remove them from the script.

Most of the database functions should not be edited because they are inserted during Capture in a way specific to the application running from the PC to the SUT. If you change these functions from when they were captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

However, simple edits can be made to some of the `Dbset()` options and more complex edits can be made for the `Data()` and `Parse()` functions to enhance multi-user emulations.

Because all users would not make the same entries to the database, you may edit the `Data()` and `Parse()` functions to vary the data that is inserted into the database during multi-user emulations for a more realistic load. You can edit the `Data()` functions in a set of scripts so that each emulated user inserts a different record.

The `Dbset()` option, `FETCHSIZE`, can be changed to see how a larger or smaller `FETCHSIZE` value affects the performance of your SUT during script execution. Please note that the `FETCHSIZE` value can not be larger than the `MAXARRSIZE` value.

7.1.5 Comments

As with all C language programs, your script file should include comments to indicate the structure of the script. Comments also are useful for interpreting the log file.

Comments are entered either during Capture or when editing your script file. They are enclosed by the /* and */ characters and may span multiple lines in a script, but may not be nested.

7.2 Data Types

The following is a list of the EMPOWER/CS data types that may be listed in the database script functions, Bind() and Define(). Format lists how the data types are represented in the script as C language data types. This format does not necessarily represent how the data type is converted and stored on the database.

<u>Type</u>	<u>Format</u>
NUMBER	Internal database binary
DECIMAL	Internal database binary

Money and Date Data Types

<u>Type</u>	<u>Format</u>	<u>Example</u>
MONEY	dollars.cents	103.45
LONGMONEY	dollars.cents	103.4500
DATE	day/month/year hour:minute:seconds	02/10/95 10:06:35
LONGDATE	day/month/year hour:minute:seconds.milliseconds	02/10/95 10:06:35.23

The format for the DATE data type must be 17 characters in length. The format for the LONGDATE data type must be 20 characters in length.

Numeric Data Types

<u>Type</u>	<u>Format</u>
BYTE	char
SHORT	short
INT	int
UINT	unsigned int
DOUBLE	double
FLOAT	float
BOOL	char

For the remaining data types, the format specifies the maximum length for the character array.

Non null-terminated characters

<u>Type</u>	<u>Format</u>
LONGCHAR	char(2 ³¹ -5)
CHAR	char(2000)
MEDCHAR	char(65535)
SHORTCHAR	char(255)

Null-terminated strings

<u>Type</u>	<u>Format</u>
LONGSTRING	char(2 ³¹ -1)
STRING	char(2001-1)
SHORTSTRING	char(256-1)

Binary data

<u>Type</u>	<u>Format</u>
LONGBINARY	char(2 ³¹ -1)
SHORTBINARY	char(256)
BINARY	char(65535)
IMAGE	char(2 ³¹ -5)

Encrypted data

<u>Type</u>	<u>Format</u>
SECURITY1	char()
SECURITY2	char(2000)

7.3 Script Arguments

Arguments can be passed to the script when the script is executed at the EMPOWER/CS command line or when a set of scripts is executed with the Multi-User testing tool, Mix. Arguments deliver information to a script to control how the script executes.

Arguments may be used to send a random number seed to a script to control a random sequence of commands. You can pass the same seed value to a script to obtain the same sequence of commands.

Arguments also may be used to control the execution of functions in the script. You may have designed the script to contain functions that test different parts of your application. The argument can specify which functions execute at what time and in what sequence.

7.3.1 Argument Syntax

The script receives arguments as parameters when the script executes. Arguments passed to the script (which do not include options) are stored as variables in the `argv[]` array. The number of elements in the `argv[]` array is defined by the variable `argc`.

EMPOWER/CS will define these argument variables automatically as follows:

```
int argc;  
char *argv[]
```

These variables can be used subsequently in the script. The number of arguments (and therefore the number of elements in the array) is limited only by your UNIX script driver which generally imposes no practical limitation on script execution.

When a script is executed, the following variables are defined:

```
argv[0] = the executable script name
argv[1] = the log file name
```

Arguments specified after the log file name in the script command are stored as `argv[2]`, `argv[3]`, and so on. You must specify a log file name if you want to specify other arguments. If you specify other arguments without specifying the log file, EMPOWER/CS will use the first argument as the log file name, thus creating an error.

7.3.2 Argument Examples

The following example illustrates a script execution:

```
$ script1 -d vagrant log1 1 3 wendy password1
```

`script1` is the name of the binary script created by the Csccl tool. The option `-d` and `vagrant` specify that the activities will be displayed on the PC named `vagrant`. `log1` is the name of the log file that will be created when the script is executed. The parameters `1`, `3`, `wendy`, and `password1` are arguments that will be accessed during script execution. `1` and `3` will be used as arguments for EMPOWER/CS functions; `wendy` will be used as the log in ID; and `password1` will be the emulated user's password.

These arguments are accessed by the script through the variables `argc` and `argv`. The values of the variables are:

```
argc = 6
argv[0] = script1
argv[1] = log1.1
argv[2] = 1
argv[3] = 3
argv[4] = wendy
argv[5] = password1
```


The source script file (the .c file) is edited to use these variables as in the following example:

```
Thinkuniform(atoi(argv[2]), atoi(argv[3]));  
Seed(atoi(argv[3]));
```

This example shows how the think time is varied in the script. The values of the variables `argv[2]` and `argv[3]` represent the limits of the think time distribution. The value of `argv[3]` also is used as the seed of the random number generator used to generate the sequence of think times.

The variable array `argv[]` contains character string variables. Since the EMPOWER/CS functions `Thinkuniform()` and `Seed()` take integer arguments rather than character strings, you must use the C library function `atoi()` to convert the character strings to integers.

The resulting log file entries for the think time functions will be:

```
Thinkuniform(1.000, 3.000)  
Seed(3.000)
```

7.4 Script Variables

EMPOWER/CS scripts are C language programs and therefore, you may use standard C variables in the script source file. All variables used in a script must be defined at the beginning of the script with the exception of `argc` and `argv`, which are defined automatically as described in the previous section.

int	-	integer
char	-	character
float	-	single precision floating point
double	-	double precision floating point

```
int i, j, k;
int i_array[10];
char c;
char product[20];
char *product_ptr;
float weight;
double iq;
```

After variables are defined, they may be assigned values:

```
i = 10;
i_array[3] = 17
c = 'a';
iq = 17.325;
product_ptr = "072148";
```

```
int i = 11
```

EMPOWER/CS-V1.0.1

7.5 Editing Your Scripts

After you have created a script with Capture, you likely will want to edit the script to change or add script functions. Of course, the changes you make to your scripts are driven by your testing goals but EMPOWER/CS scripts generally are edited to achieve the exact emulation environment desired. Because EMPOWER/CS scripts are actually C language programs, they can be edited to include branching and looping. Special EMPOWER/CS functions can be added to your scripts for advanced control of script execution. The following sections describe EMPOWER/CS functions that can be inserted into your scripts only by editing, and also describe methods for branching and looping.

7.5.1 Recording Messages

The `Log()` function records a message in the executed script's log file. `Log()` functions are similar to comments in that they provide help in following the structure of a script. The parameter of the `Log()` function must be a character string.

The `Note()` function is used to place a note in the "Note" column of the EMPOWER/CS Monitor View 5 during script execution. The parameter of `Note()` is a character string which is saved in the log file and subsequently displayed by Monitor.

The `Inote()` function specifies a message that also will appear in the "Note" column of Monitor View 5. The parameter of `Inote()` is an integer that will be displayed by Monitor.

You can insert these functions into your script as needed when you edit your script file.

7.5.2 File Input/Output Functions

Creating scripts that read input files on the UNIX script driver is a common practice during load testing. Such input files may contain names, addresses, phone numbers, etc., that are used for emulating database queries and updates. EMPOWER/CS allows you to have a separate input file for each emulated user, or multiple users can share a single input file. Using a single input file is often more convenient since you do not have to maintain many files for large tests.

EMPOWER/CS provides a convenient way for scripts to read from an input file, ensuring that data for each script is unique. The File I/O functions are used in scripts to read and write files which is useful for load tests requiring interaction with data files on the UNIX script driver machine. These capabilities also are useful in simplifying complex scripts such as database entry scripts. The EMPOWER/CS file input/output functions allow you to read data from a file, send data from the file to the SUT, receive data from the SUT, and write those data to a file. These functions simplify the C language statements that would need to be added to accomplish the same thing. You can insert File Input/Output functions when editing your script.

The environment variable `E_FIOPATH` can be used to specify the directory in which the files to be accessed reside. A file must be opened before it can be accessed with the file input/output functions. `Fioopen` may be used to open a file. Also, the `Fioreadline`, `Fioreadfield`, `Fioreadchar`, and `Fiowritechar` functions automatically open the file before reading from or writing to it. The function `Fioshare` is used to identify a file that is to be shared. If a file contains NULL characters, an error will occur when the file is read by an input/output function.

Three global variables are used for file input/output. They are defined automatically as follows:

```
unsigned char *FIOBUFFER
int FIOLEN
int FIOBUFFERSZ
```

The variable `FIOBUFFER` is a pointer to the characters read from the file. This variable often is used when sending data read from a file to the SUT. The variable `FIOLEN` is the number of valid characters in `FIOBUFFER`. If the value of `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached. The variable `FIOBUFFERSZ` is the maximum size of the data that can be read at one time. The default value of `FIOBUFFERSZ` is 512 characters. If the value of `FIOBUFFERSZ` is redefined in a script, it must be redefined before any file input/output functions that reference the file are encountered.

The file input/output functions are:

<code>Fioshare</code>	<code>Fioreadline</code>	<code>Fioskipline</code>	<code>Fioseek</code>
<code>Fiounshare</code>	<code>Fioreadfield</code>	<code>Fioskipfield</code>	<code>Fioautorewind</code>
<code>Fioopen</code>	<code>Fioreadfields</code>	<code>Fioskipchar</code>	<code>Fiorewind</code>
<code>Fioclose</code>	<code>Fioreadchar</code>		
<code>Fiodelimiter</code>	<code>Fiowritechar</code>		

These functions are described in the following sections.

7.5.2.1 Fioshare

The syntax for this function is:

```
Fioshare(filename)
```

The `Fioshare()` function identifies a file that is to be shared. It must be called before any other File I/O functions are called to reference the same file. `Fioshare` in a script presumes execution of the `fioshare` command at the UNIX script driver's shell prompt. The `fioshare` command creates a global variable that contains the offset for the next byte to be read from a shared file. The value of the variable (offset) remains between tests, so you can continue to read an input file from the point left by the previous test. This saving of the offset is useful in tests that corrupt a database on the server. The ability to avoid the same transactions means you can avoid restoring the database before every test. You must execute the `fioshare` command if you want to resume reading from the beginning of the

input file. For this reason, `fioshare` is often run from Mix command files that set up for a new test.

For example, assume we require scripts to read commands from an input file called `cmds`. The following command will create the global offset for the file:

```
$ fioshare cmds
```

A segment in the script to read from `cmds` might look like the following. Each instance of this script will read different lines.

```
Fioshare("cmds");  
  
Fioreadline("cmds");  
if (FIOLEN == -1)  
{  
    Log("end of the date file");  
    exit(1);  
}  
Type(FIOBUFFER, "^M", "");
```

The global offset is stored in an EMPOWER/CS global variable. (Refer to the Multi-User Testing Manual for more information on global variables.) This global variable's name is the inode of the shared file. This can be confirmed by typing the UNIX `"ls -i"` command with an argument of the shared file and then running the `gv_stat` command, which lists the names, status, and value of EMPOWER/CS global variables. For example:

```
$ fioshare cmds  
$ ls -i cmds  
59449 cmds  
$ gv_stat  
gv_stat:  EMPOWER/GV V1.0.0, Serial#R00000-000, Copyright PERFORMIX,  
Inc. 1988-95  
Name      Type      Value Allocated Protector  
-----  
59449     long int    0         0
```


end for reading and writing. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.4 Fioclose

The syntax for this function is:

Fioclose(filename)

The `Fileclose()` function closes the file `filename`. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.5 Fiodelimiter

The syntax for this function is:

```
Fiodelimiter(filename, delimiters)
```

The `Fiodelimiter()` function defines the field delimiters for the file `filename`. The default is `"\t\n"` and `"\n"` is always a delimiter ("`\t`" is tab and "`\n`" is new line or linefeed). If the function is successful, zero is returned. If an error occurs, `-1` is returned.

7.5.2.6 Fioreadline

The syntax for this function is:

```
Fioreadline(filename)
```


7.5.2.7 Fioreadfield

7.5.2.8 Fioreadfields

7.5.2.9 Fioreadchar

The syntax for this function is:

```
Fioreadchar(filename, n)
```

The `Fioreadchar()` function reads `n` bytes from the file `filename`. If the file is not currently open (see `Fioopen`), it is opened by `Fioreadchar()`. A pointer to `FIOBUFFER` is returned. If `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached.

7.5.2.10 Fiowritechar

The syntax for this function is:

```
Fiowritechar(filename, buf, n)
```

This function writes `n` bytes from the buffer, `buf`, to the file `filename`. If the file is not currently open (see `Fioopen`), it is created or truncated and opened for reading and writing automatically. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.11 Fioskipline

The syntax for this function is:

```
Fioskipline(filename, n)
```

The `Fioskipline()` function skips forward `n` lines in the file `filename`. If the file is not currently open (see `Fioopen`), it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last line read. If the function is successful, zero is returned. If an error occurs, -1 is returned.

The `Fioskipfield()` function skips forward `n` fields in the file `filename`. The fields are separated by the delimiter (see `Fiodelimiter`). If the file is not currently open (see `Fioopen`), it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last field read. If the function is successful, zero is returned. If an error occurs, -1 is returned.

```
Fioskipchar(filename, n)
```

The `Fioskipchar()` function skips forward `n` characters in the file `filename`. If the file is not currently open (see `Fioopen`), it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last characters read (the number of characters used to update `FIOBUFFER` and `FIOLEN` is defined by the variable `FIOBUFFERSZ`). If the function is successful, zero is returned. If an error occurs, -1 is returned.

`Fioseek(filename, offset)`

The `Fioseek()` function sets the file pointer to a specific byte in the file. The next byte read or written will occur at `offset` bytes from the beginning of the file. If the value of `offset` is equal to `FIOEND`, the seek will occur to the end of the file. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.15 Fiorewind

The syntax for this function is:

```
Fiorewind(filename)
```

The `Fiorewind()` function rewinds the file pointer to the beginning of the file for the file `filename`. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.16 Fioautorewind

The syntax for this function is:

Fioautorewind(filename)

The `Fioautorewind()` function automatically rewinds the file pointer to the beginning of the file specified in `filename`, whenever the end-of-file is encountered. This function is useful if multiple scripts read data from one file.

7.5.2.17 File Input/Output Function Examples

The following example illustrates using the functions `Fioreadfield()` and `Fioreadfields()` to read one or more fields from a file. `Fiodelimiter()` is used to specify that the field delimiter in the file `inputfile` is a comma.

```
char last[20];
char first[20];

...

Fioopen("inputfile", "r");
Fiodelimiter("inputfile", ",");
Fioreadfield("inputfile");
Type("%s", FIOBUFFER);

LeftButtonPress(360,211);

AppWait(0.06);
WindowRcv("SfSfPt");

Fioreadfields("inputfile", 2, last, first);
Type("%s", last);

LeftButtonPress(357,252);

AppWait(0.06);
WindowRcv("SfSfPt");

Type("%s", first);
```

The following is a portion of the file `inputfile`:

312890463 doe, jane 294028190 smith, john
--

```
Fioreadline("info");
if (FIOLEN== -1){
Fiorewind("info");
Fioreadline("info");
}
Type("%s", FIOBUFFER);
```

7.5.3 Pacing Functions

EMPOWER/CS allows you to insert three functions into your script file for controlling the pace of the script. These pacing functions cause the script to pause long enough to make the script send transactions to the SUT at a predetermined throughput. Typically used in a loop, these functions may be nested to permit controlled throughput of transactions within a larger transaction.

Each of the pacing functions accepts an argument naming the pace and defining the speed of the pace.

`Paceconstant()` causes transactions to be submitted at a constant throughput. The second argument to `Paceconstant()` defines the number of seconds that the script should take since the last call to `Paceconstant()`. The first call to a pacing function does not delay; it is used as a starting point for the subsequent calls. For

this reason, the first call to a pacing function is often made with arguments of 0 to define the speed.

`Paceuniform()` and `Pacetne()` control throughput but do not do so at a constant rate. The functions have an average throughput that will be maintained, but submission of transactions occurs at frequencies other than that defined by a constant distribution.

`Paceuniform()` accepts two arguments to identify the speed of the pace - a minimum delay and a maximum delay. The average of the two will be maintained during sustained execution of the script. Each time `Paceuniform()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a uniform distribution between the minimum and maximum values. For example, if `Paceuniform("query", 8.0, 12.0)` is used in a script, a sequence of 9, 11, 8, 10, and 12 second delays is possible (assuming the query has a response time of zero seconds.) The average of the delays is still ten seconds. `Paceuniform()` will select the values for delay accurate to 1/100th of a second, so 9.27 seconds is a possible value.

`Pacetne()` accepts three arguments to identify the speed of the pace - a minimum, average and a maximum delay. The average will be maintained during sustained execution of the script. Each time `Pacetne()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a truncated, negative exponential distribution defined by the three values. In a typical such distribution, the average is relatively close to the minimum. For example, if `Pacetne("query", 7.0, 10.0, 20.0)` is used in a script, a sequence of 7, 8, 10, and 15 second delays is possible. The average of the delays is still ten seconds. `Pacetne()` will select the values for delay accurate to 1/10th of a second, so 9.2 seconds is a possible value.

7.5.4 Advanced Use of Script Variables

Five EMPOWER/CS functions exist that allow you to manipulate script variables. They are `CmpVar()`, `GetVar()`, `GetIntVar()`, `SetVar()`, `SetIntVar()`. These functions allow you to test, update, read, and compare variables.

You may insert the `CmpVar()` function into your script to compare a variable from a SQL statement to a specified value. The syntax for this function is:

```
CmpVar(curnum, var, value);
```

This function determines if the specified variable, `var`, in the current row is equal to the value specified in the `value` parameter. The address of the variable must be passed to `CmpVar()`.

This function could be used for fetching records until a certain value is returned. An example of this function follows:

```
Parse(CUR1, "select ename from employee_table");

Define(CUR1, "1", STRING, 50);

Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 1);
while (CmpVar(CUR1, "1", "Smith") != 0){
    Fetch(CUR1);
    GetNextRow(CUR1);
}
```

You can insert the `GetVar()` or `GetIntVar()` functions into your script to return the current value of a specified variable (either the name or the position) in the SQL statement. `GetVar()` is used for string variables and `GetIntVar()` is used for integer variables. These functions should be inserted into the script after the `Fetch()` and `GetNextRow()` functions.

The following example demonstrates using the `GetVar()` function within a script:

```
char *empname, *empno;

...

Parse(CUR1, "select ename, empno from employee_table");

Define(CUR1, "1", STRING, 50);
Define(CUR1, "2", INT, 4);

Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 1);
while (Fetch(CUR1) != 0){
    GetNextRow(CUR1);
    empname=GetVar(CUR1, "1");
    printf("empname is %s\n", empname);
    empno=GetVar(CUR1, "2");
    printf("empno is %d\n", *(int *)empno);
}
```

The `SetVar()` and `SetIntVar()` functions assign a new value to a specified variable. The syntax for these functions follows:

```
SetVar(curnum, var, value);
SetIntVar(curnum, var, value);
```

The new value for the variable, `var`, is assigned in the parameter value. These functions could be inserted into the script before an `Exec()` function or after the `GetNextRow()` or `GetVar()` functions.

The following example demonstrates using `SetIntVar()` in a script:

```

int empno,deptno,i;

...

Parse(CUR1, "select ename from employee_table where empno=:empno
and deptno=:deptno");

Bindp(CUR1, 1, INT, 4);          /* pos 1 is :empno */
Bindp(CUR1, 2, INT, 4);          /* pos 2 is :deptno */

Data(CUR1, "23|20");

empno=GetIntVar(CUR1, "1");
deptno=GetIntVar(CUR1, "2");

/* keep executing above parse statement changing the empno
everytime */
for (i=0;i<5;i++){
    empno=empno+i;
    SetIntVar(CUR1, "2", empno);

    Exec(CUR1);

    if (Fetch(CUR1) != 0)
        printf("empno %d is in deptno %d\n", empno, deptno);
    else
        printf("empno %d is not in deptno %d\n", empno,
deptno);
}

```

You must insert loops very carefully if you plan to execute scripts in both Non-Display and Display modes. You must be sure to encompass all relevant functions in the loop for Non-Display and Display script execution. For example, you must include all relevant button presses within the loop so that the correct sequence of buttons will be activated during script execution in Display mode.

Activate the **XY** pushbutton. The following window appears displaying the current xy coordinates of your mouse:

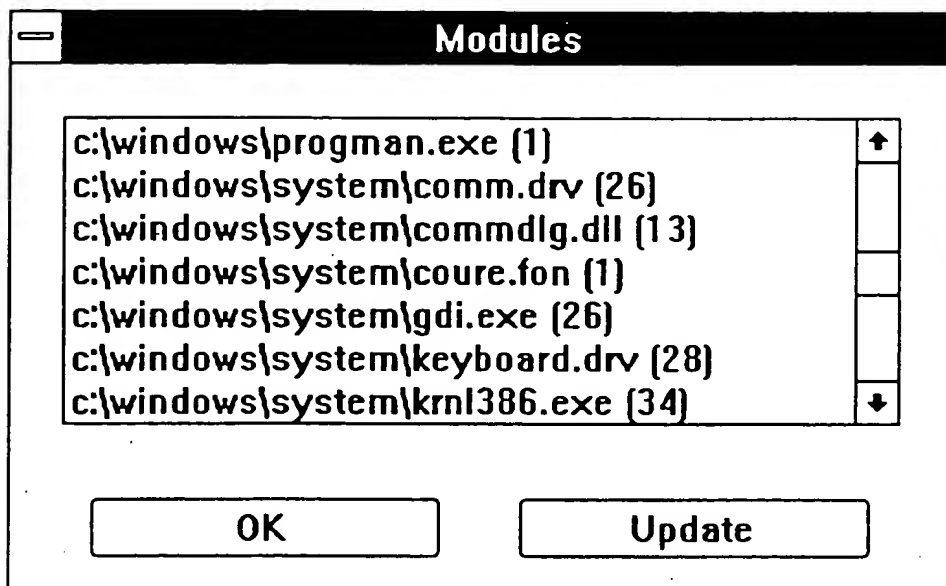
= XY	
<u>F</u> ile	<u>H</u> elp
(99,134)	
Trace <input checked="" type="checkbox"/>	

The **Trace** option of this tool follows your mouse movements listing the on screen xy coordinates as the mouse moves. If you de-select **Trace**, you can type a number in the left field that will move the mouse to that x position on screen. If you tab to the next field, you can enter a y position that will move the mouse to that y position. As you type each number, the mouse will move to the specified position.

Select **Exit** from the **File** menu to close this tool.

8.2 Modules

Select the **Modules** button to activate the following window:



This tool alphabetically lists all the processes currently running in Windows. The number listed in parentheses indicates how many instances of the process are running. **Update** updates all process information.

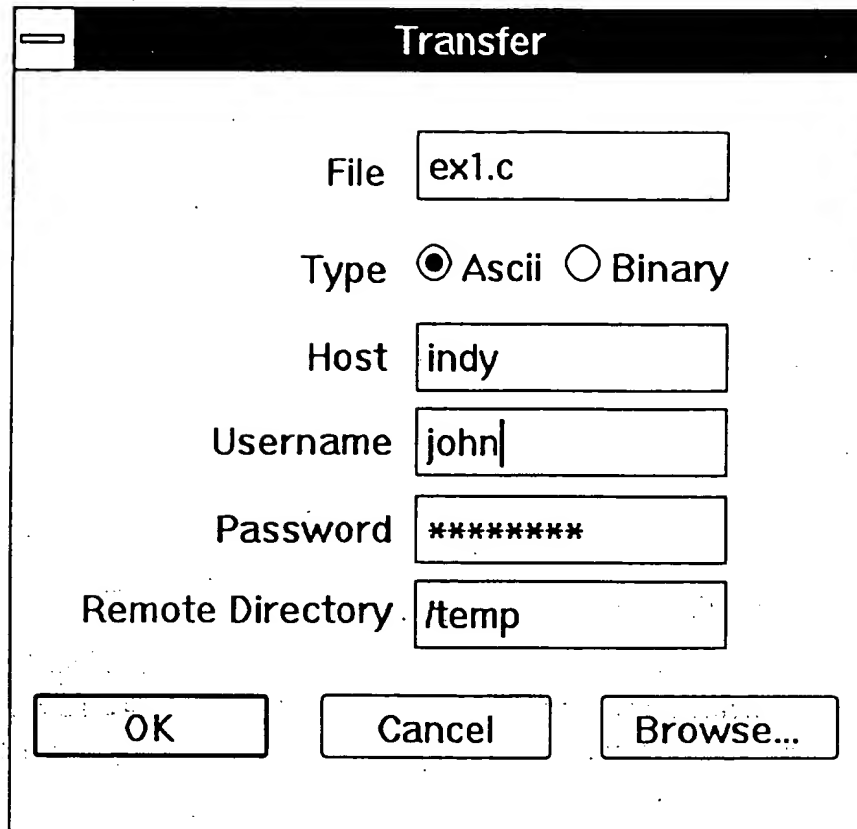
The Modules tool can be used for debugging. For example, if you encountered an error when trying to Capture client/server activity, you could use Modules to verify that winsock.dll is installed on your PC.

Select **OK** to exit this tool.

8.3 Transfer

The Transfer tool allows you to transfer files manually from the PC to the UNIX script driver.

Select the Transfer pushbutton in the Tools window. The following window will open:



A dialog box titled "Transfer" with a standard window control bar (minimize, maximize, close buttons). The dialog contains several input fields and radio buttons. The "File" field contains "ex1.c". The "Type" field has two radio buttons: "Ascii" (selected) and "Binary". The "Host" field contains "indy". The "Username" field contains "john". The "Password" field contains "*****". The "Remote Directory" field contains "/temp". At the bottom, there are three buttons: "OK", "Cancel", and "Browse...".

Transfer	
File	ex1.c
Type	<input checked="" type="radio"/> Ascii <input type="radio"/> Binary
Host	indy
Username	john
Password	*****
Remote Directory	/temp
OK Cancel Browse...	

Enter the complete name of the script or data file you wish to transfer and specify in what form you wish to transfer the file, either as **ASCII** or **Binary**.

In DOS files, an end-of-line includes a carriage return and a linefeed whereas in UNIX an end-of-line requires only a linefeed. Transferring a file as **ASCII** translates the carriage return/linefeed combinations in a DOS file to the single linefeed required in UNIX.

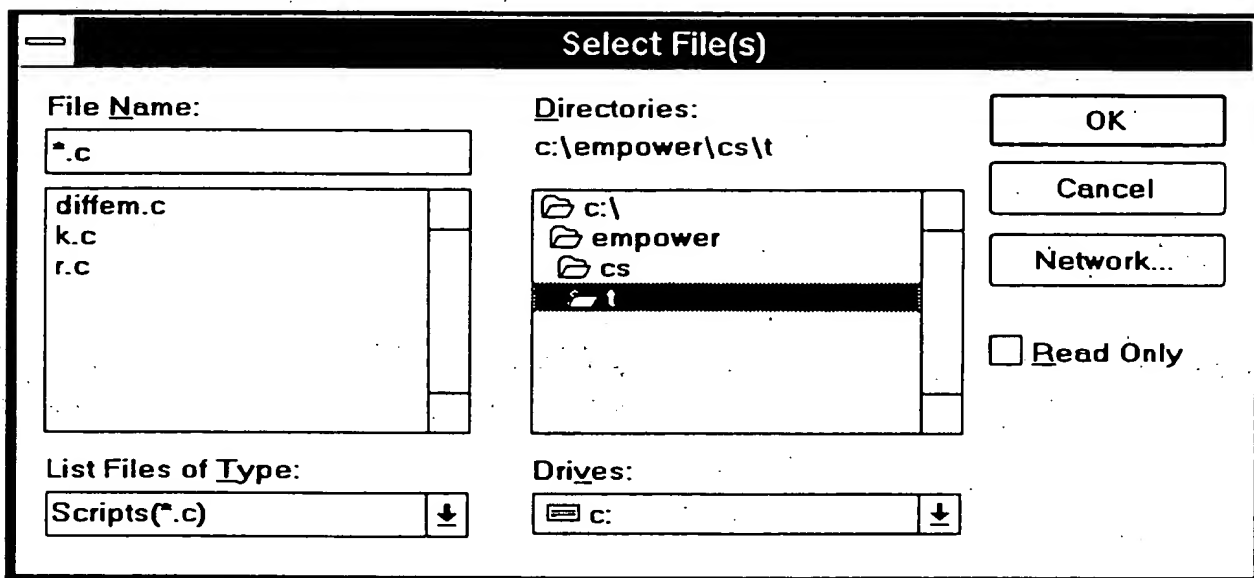
If you specify **Binary**, the file is transferred without character translations. Therefore, you should specify **Binary** for transferring files in which data can not be altered and all characters are required for the file (i.e., images).

If large amounts of data (i.e., images, large text files, etc.) are input to the database during Capture, such data is inserted into separate data files. If you are transferring

a script that has data files, you must transfer the associated data files separately in Binary mode. Scripts should be transferred as ASCII files.

If you need to locate a file, select the **Browse** pushbutton in the **Transfer** window to browse all available directories.

The following **Browse** window will open:

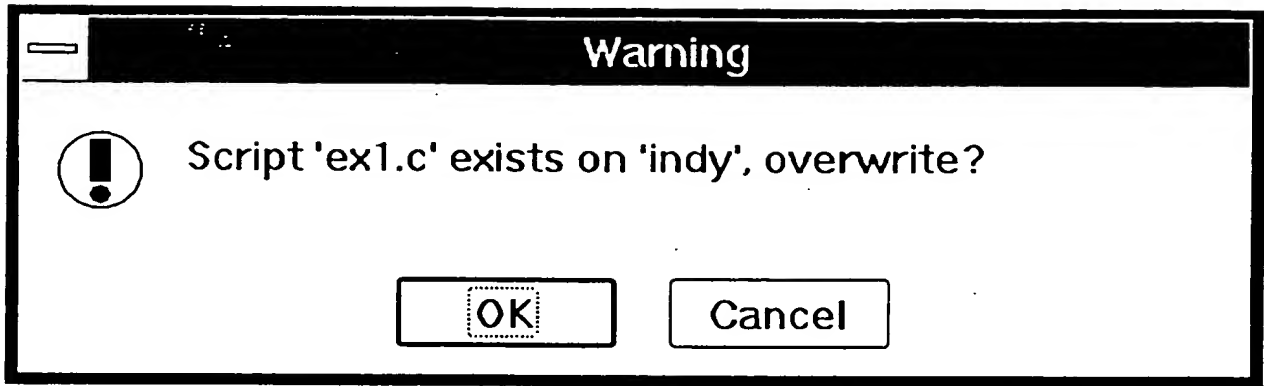


In this window, you may specify or search through all available directories, files, and disk drives. You may select multiple files from a specific directory. When you have located all needed files, choose **OK** or **Cancel** as appropriate to return to the **Transfer** window.

After specifying a file, you should enter the **Host** name (the UNIX driver), the appropriate **Username** and **Password**, and the **Remote Directory** (on the UNIX driver) where you wish to transfer the script file.

In the **Transfer** window, select **OK** to transfer the specified file to the UNIX driver machine.

If the script .c file already exists on the UNIX driver, you will receive the following message:



Choose OK or Cancel as appropriate.

8.4 Tree

The Tree tool allows you to list the tree structure for a MS Windows pushbutton, such as OK, Cancel, Yes, No, etc.

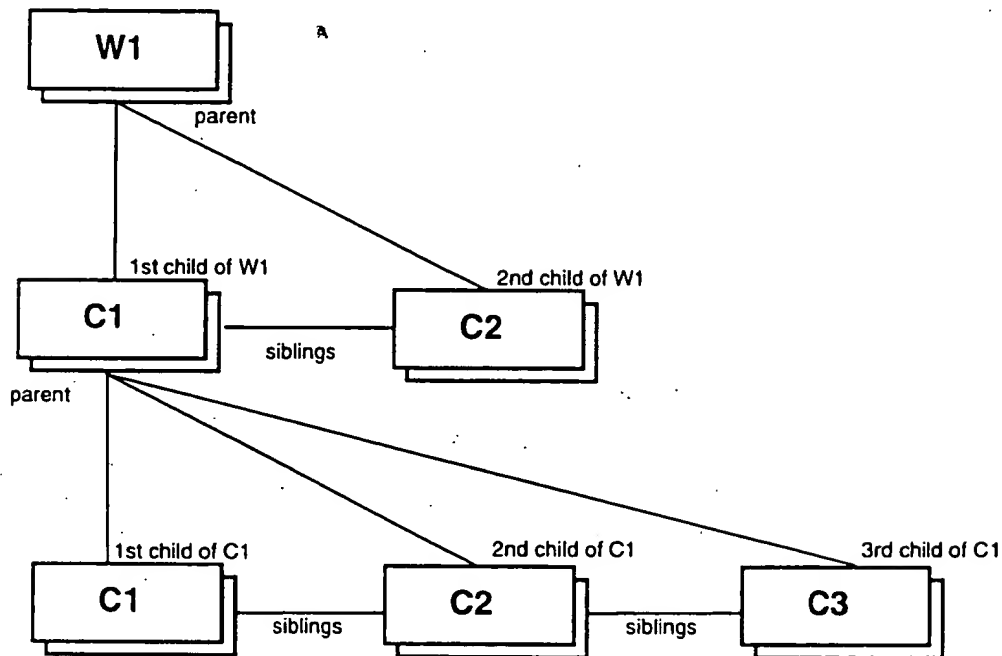
This tool can be used if you wish to manually insert or change a `ButtonPush()` function. It allows you to determine the tree structure for the `ButtonPush()` function's `str` parameter.

8.4.1 The Tree Structure

The MS Windows tree structure is based on a heirarchy of windows where each window that is accessed from a primary, or parent window, is a child of that parent. For instance, each window or pushbutton that is activated from another window is a child of the window it opens from. If two or more equal level windows are

accessible from a parent window, they are children of that parent window and siblings of each other.

The following diagram demonstrates this structure:



The Tree tool lists a tree structure from right to left, from child to parent, where the right-most item is the name of the button or window you selected. If one of the windows or the pushbutton has no title, something like "#c1" will be listed to indicate the window or pushbutton is a certain numbered child of the preceding parent window.

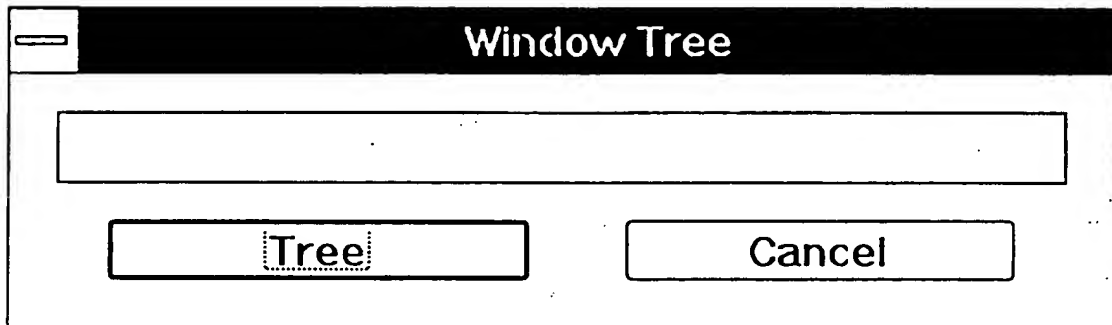
Note: In some rare cases, applications are designed where a window requires its sibling window to also be its parent in which case something like "#s1" may be listed in the structure.

A tree structure from the above diagram could look like: W1|#c1|#c2. In this example, #c2 is the second child of the first child (#c1) of the window W1.

In the following tree structure, Tools|WindowTree|Cancel, Cancel is the button contained in the window WindowTree which is a child of the Tools window.

8.4.2 Using the Tree Tool

Select the Tree button from the Tools window which opens the following window:



Activate the Tree button. The mouse pointer will turn into a large arrow.

With this new pointer, select the button for which you wish to list the Tree structure. This tree structure is listed in a particular hierarchy as explained above in Section 8.4.1.

As an example, obtain the Tree structure for the Cancel button, by selecting Tree. Then, select Cancel in the Window Tree window as shown below:

The following structure will appear in the Window Tree window:

Tools|Window Tree|Cancel

When you have obtained all needed tree structures, select the Cancel button or Close from the Window Tree window's System menu.

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☒ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER: _____**

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.